공학박사 학위논문

비대칭 멀티코어 아키텍처용 공정성 보장 및 에너지 효율적인 스케줄링

Fair-share and Energy-efficient Scheduling in Performance-asymmetric Multicore Architecture

2016년 2 월

서울대학교 대학원

전기 · 컴퓨터공학부

김 명 선

비대칭 멀티코어 아키텍처용 공정성 보장 및 에너지 효율적인 스케줄링

Fair-share and Energy-efficient Scheduling in Performance-asymmetric Multicore Architecture

지도 교수 홍 성 수

이 논문을 공학박사 학위논문으로 제출함

2016 년 2 월

서울대학교 대학원
전기·컴퓨터 공학부
김 명 선

김 명 선의 공학박사 학위논문을 인준함
2016 년 2 월

위 원 장 : 김 태 환 (인)
부위원장 : 홍 성 수 (인)
위 원 : 심 규 석 (인)
위 원 : 엄 현 상 (인)
위 원 : 전 광 일 (인)

초 록

최근 우수한 사용자 체감과 질적으로 향상된 수준의 서비스를 위하여 스마트폰, 태블릿과 같은 임베디드 시스템에서 비대칭 멀티코어 아키텍처의 사용이 크게 증가되고 있다. 이는, 이러한 아키텍처가 임베디드 시스템의 제한된 면적과 소비전력 환경하에서 주는 하드웨어적 이점 때문이다. 임베디드 시스템에 사용되는 비대칭 멀티코어 아키텍처는 서로 다른 특성을 가지는 두 가지 타입의 코어들로 이루어진다. 첫 번째 타입의 코어는 높은 성능과 낮은 에너지 효율성을 특징으로 하고, 두 번째 타입의 코어는 낮은 성능 대신 높은 에너지 효율성을 특징으로 한다.

비대칭 멀티코어 아키텍처를 운용하는 방법에는, 각 코어들을 사용하는 방법에 따라서 두 가지 종류가 있다. (1) 한 개의 높은 성능 코어와 한 개의 에너지 효율적인 코어를 하나의 pair로 형성한 후, 그 중 한 개의 코어만 동작하게 하는 코어 타입 선택 방식과 (2) 시스템의 모든 코어를 동시에 사용할 수 있는 전체 코어 사용 방식으로 운용된다. Linux kernel은 이러한 비대칭 멀티코어 아키텍처에 가장 널리 쓰이는 운영체제이며, CFS(completely fair scheduler)를 사용하여 태스크들을 스케줄링 한다. 또한, CFS는 비대칭 멀티코어 아키텍처의 코어 타입 선택 방식과 전체 코어 사용 방식에 맞는 스케줄링 프레임웍을 제공하고 있다.

하지만 현재 제공되고 있는 두 가지 코어 사용 방식을 위한 스케줄링 프레임웍에는 다음과 같은 문제점이 있다. 첫째, 코어 타입 선택 방식에서의 스케줄링 프레임웍은 비대칭 멀티코어간 부하분산 시

태스크의 가중치(weight)만 고려하여 부하분산을 수행한다. 이로 인하여 필요이상으로 코어의 동작 주파수를 상승시키거나, 에너지 효율적인 코어에서 수행해도 충분한 태스크가 고성능 코어에서 운용되어 소비전력을 크게 증가시키는 문제점을 야기시킨다.

둘째, Linux kernel의 CFS는 virtual runtime을 통하여, 태스크들에게 가중치에 비례하는 CPU 사용 시간을 부여한다. 이때, 태스크의 virtual runtime 산정 시, 현재 태스크를 수행하는 코어의 상태(코어 타입 혹은 동작 주파수 등)를 고려하지 않는다. 이로 인하여 공정한 CPU 사용 시간을 태스크들에게 부여하지 못한다. 또한, CFS는 태스크들의 virtual runtime을 동일하게 만들려고 노력한다. 이는 태스크들의 상대적인 수행 정도를 비슷하게 유지시키기 위해서이다. 하지만 이는 개별 코어에서는 유지되나, 시스템 전체적으로 비슷하게 유지 되지 않는다. 이는 시간이 지남에 따라서 태스크간 virtual runtime 차이를 증가시켜, 태스크간 상대적인 수행 정도 차이가 더욱 더 커지게 하는 문제점을 발생시킨다.

본 학위논문은 비대칭 멀티코어 아키텍처가 지원하는 코어 타입 선택 방식과 전체 코어 사용 방식에 최적화된 스케줄링 기법을 제안한다. 첫째, 본 연구는 코어 타입 선택 방식의 저전력 운용 목적에 맞는 스케줄링 기법을 제안한다. 이를 위하여 먼저, Linux kernel이 비대칭 멀티코어 아키텍처를 위하여 제공하는 DVFS(Dynamic Voltage and Frequency Scaling) 정책을 정확히 분석한다. 분석된 결과를 토대로 정확한 동작을 모델링하고, 이를 제안하는 스케줄링 기법에 반영한다. 이 기법은 부하분산 시 태스크의 가중치뿐만 아니라, 코어의 사용률을 고려하여 부하를 분산시킨다. 이를 통하여 성능 저하를 최소화 하면서

주파수 상승을 억제하고, 동시에 고성능 코어를 최대한 적게 사용하는 저전력, 에너지 효율적인 스케줄링을 수행한다.

둘째, 본 연구는 전체 코어 사용 방식에 적합한 공정할당 스케줄링 방식을 제안한다. 이는 코어의 상태를 반영한 스케일된 CPU 시간을 구한다. 이를 CFS의 virtual runtime에 반영하고, SVR (scaled virtual runtime)로 확장한다. 또한 각각의 고성능 코어로 이루어진 클러스터와 에너지 효율적인 코어로 이루어진 클러스터 내부에서, 모든 태스크들의 SVR 차이를 일정한 상수 크기로 제한시킨다. 이를 통하여 클러스터 내부의 모든 태스크들의 상대적 진척 정도를 비슷하게 유지시킨다. 결과적으로, 시스템 전체적인 공정할당 스케줄링을 수행하도록 한다.

본 연구에서 제안하는 두 가지 스케줄링 기법들의 효용성을 입증하기 위해서, 실제 상용으로 출시된 비대칭 멀티코어 아키텍처 기반 제품에 이들을 구현하였다. ARM사의 빅리틀 아키텍처를 대상 시스템으로 하였으며, 이는 임베디드 시스템에서 가장 대표적으로 사용되는 비대칭 멀티코어 아키텍처이다. 저전력 스케줄링 기법은 코어 타입 선택 방식이 지원되는 Galaxy S4 Android 스마트폰에 구현되었다. CPU-intensive한 부하를 사용하여 실험한 결과, 기존 대비 최대 11.35%의 에너지 소비가 감소하였다. 또한 Android 응용프로그램을 이용하여 실험 시, 기존 대비 동일한 QoS를 유지하면서 7.35% 에너지 소비가 감소하였다. 이러한 실험 결과는 비대칭 멀티코어에서 제안된 스케줄링 기법이 에너지 효율적임을 증명한다.

공정할당 스케줄링 기법은, 전체 코어 사용 방식이 지원되는 ARM사의 Versatile Express TC2 board에 구현하였다. 실험 결과,

클러스터 내부의 태스크간 SVR 차이가 상수 값 이내로 제한됨을 확인하였다. 또한 SVR 차이를 작게 만드는 것이 공정할당 스케줄링에 어떠한 영향을 실질적으로 미치는지 알기 위해서, 동일한 태스크를 여러 개 생성한 후 그들의 완료시간 표준 편차를 측정하였다. 실험 결과, 기존 대비 완료시간 표준편차가 56% 줄어 들었다. 이러한 실험 결과는 본 논문에서 제안된 스케줄링 기법이 태스크들에게 더 공정한 CPU 시간을 부여함을 직관적으로 알 수 있게 한다.

주요어: 비대칭 멀티코어, 태스크 스케줄링, 멀티코어 부하분산, 저전력 스케줄링, 빅리틀 아키텍처

학 번: 2011-30218

목 차

그림 목차

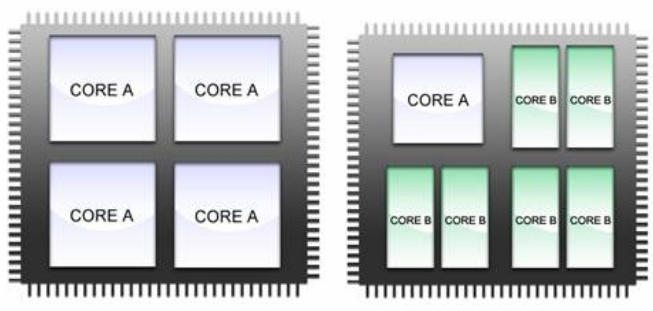
7

표 목차

제 1 장 서 론

스마트폰, 태블릿과 같은 현대의 임베디드 시스템은 복잡해짐과 동시에 LTE와 같은 고속 모뎀 통신, 다양한 센서 모듈을 통한 빠른 주변환경 인식, 고해상도 디스플레이, 대용량 멀티미디어 컨텐츠 실행 기능 등을 갖추고 있다. 이러한 경향으로 인하여, 임베디드 시스템 사용자들은 훨씬 더 향상된 사용자 체감과 높은 질의 서비스를 요구하고 있다. 이를 위하여 임베디드 시스템은 고성능 멀티코어를 사용하기 시작했다. 또한, 향상된 반도체 기술에 힘입어 갈수록 코어의 숫자는 늘어가고 코어의 동작 주파수는 높아지고 있다.

대부분의 고성능 멀티코어 아키텍처는 그림 1의 (a)와 같은 대칭 멀티코어(symmetric multicore processor)로 구성되고, 각 코어에는 수퍼스칼라(superscalar), 다 계층 파이프라인 스테이지(multi-level pipeline



(a) (b)

그림 1. 멀티코어 아키텍처: (a) 대칭 멀티코어, (b) 비대칭 멀티코어

stage), 비순차적 수행(out of order)과 같은 기술들이 내제되어 있다. 하지만 이러한 기술들은 임베디드 시스템에서 사용되는 SoC(system on a chip)에서 상대적으로 넓은 면적을 차지하며, 높은 소비전력을 나타낸다. 코어 수가 증가되면서, 이러한 특성은 SoC의 면적과 소비전력 측면에서 문제로 부각되었다 [1][5][6].

이러한 문제점을 해결하기 위해서 임베디드 시스템 개발자들은 그림 1의 (b)에 나타낸 비대칭 멀티코어(performance-asymmetric multicore processor) 아키텍처를 채택하기 시작하였다. 비대칭 멀티코어 아키텍처는 같은 명령어 집합(single-ISA: single instruction set architecture)을 사용하고 두 가지 종류의 코어로 이루어져 있다: 1) 차지하는 면적이 크고 높은 주파수로 운용되는 고성능 코어, 2) 면적이 작고 주로 낮은 주파수로 운용되는 에너지 효율적인 코어. 고성능을 필요로 하는 응용프로그램과 백그라운드 혹은 IO 집중적인 응용프로그램들을 각각 고성능 코어와 에너지 효율적인 코어에 할당함으로써, 같은 코어 수의 대칭 멀티코어 아키텍처보다 향상된 단위 전력당 성능(performance/watt)을 나타낼 수 있다.

비대칭 멀티코어 아키텍처의 동작 모드는 1) 한 개의 고성능 코어와 한 개의 에너지 효율적 코어를 pair로 구성하여 둘 중 한 개만 동작시켜서 저전력 효과를 극대화하는 코어 타입 선택 방식과 2) 모든 코어를 동시에 사용할 수 있고, 또한 모든 코어가 독립적으로 제어될 수 있는 전체 코어 사용 방식 두 가지가 있다. 이러한 두 가지 동작 모드 각각의 장점을 최대한 활용하기 위해서는 운용되는 운영체제의 스케줄링 방식이 각각의 모드에 맞게 최적화 되어야 한다. 코어 타입 선택 방식에서의 스케줄러는 코어에 태스크를 할당할 때, 고성능 코어와 에너지 효율적인 코어 중, 해당 시점에 어느 코어에 할당해야 가장 저전력 효과가 있는지를 알

아야 한다. 전체 코어 사용 방식에서의 스케줄러는 멀티코어를 사용함에 있어서, 각 태스크에게 공정성을 보장해야 한다. 즉, 스케줄러는 태스크를 코어에 할당할 때, 공정하게 고성능 코어와 에너지 효율적인 코어를 태스크들이 사용할 수 있도록 설계되어야 한다.

본 논문에서는 이러한 비대칭 멀티코어 아키텍처의 두 가지 동작모드에 최적화된 스케줄링 알고리즘을 제안한다. 또한, 제안된 기법들의 성능을 해당 동작 모드에 적용해 그 실효성과 효과를 검증한다. 본 장의 1절에서는 비대칭 멀티코어 아키텍처의 두 가지 동작모드에 대하여 기존의 스케줄링 방식이 가지고 있는 문제점 제시 및 본 연구에 대한 동기를 기술한다. 이어서, 2절에서는 본 연구에서 제시한 알고리즘들의 기여에 대하여 논한다. 마지막으로, 3절은 본 논문 전체의 구성을 설명한다.

제 1 절 연구 동기

고성능 멀티미디어 컨텐츠 프로세싱 기능 및 빠른 사용자 응답 특성 등을 임베디드 시스템이 가지는 제한된 소비전력 환경하에 수행하려면, 비대칭 멀티코어 아키텍처가 가지는 하드웨어적 특성을 최대한 인지한 스케줄링 기법이 필요하다. 구체적으로, 본 연구에서는 비대칭 멀티코어 아키텍처의 두 가지 동작모드에서, 현재 사용되는 스케줄링 방식의 문제점을 설명하고, 각각의 문제점에 대한 해결책을 제시한다. 본 연구논문의 목적은 비대칭 멀티코어 아키텍처의 두 가지 동작 모드에 최적화된 스케줄링 알고리즘들을 제안하고, 각각의 동작 모드에서 제안된 알고리즘의 실효성을 증명하는 것이다. 비대칭 멀티코어 아키텍처의 두 가지 동작모드에 대한 스케줄링 알고리즘을 연구하게 된 동기를

설명하면 다음과 같다.

1.1 코어 타입 선택 방식에서의 저전력 스케줄링 최적화

비대칭 멀티코어 아키텍처가 코어 타입 선택 방식에서 동작할 때, 각 pair에 존재하는 고성능 코어와 에너지 효율적인 코어 중 한 개의 코어만 사용된다. 이때, 사용되는 코어의 결정은 현재 사용되는 코어의 사용 정도에 따라 결정된다. 코어의 사용률이 어느 한계 보다 높을 때는 현재 주파수보다 더 높은 주파수로 변경하거나, 혹은 에너지 효율적인 코어에서 고성능 코어로 스위칭된다. 반대로 어느 한계보다 낮을 때는 현재 주파수보다 낮은 주파수로 변경하거나, 혹은 고성능 코어에서 에너지 효율적인 코어로 스위칭된다. 따라서 스케줄러가 태스크를 어떤 코어에 할당하는지에 따라 코어가 스위칭되거나 혹은 동작 주파수의 변경이 생긴다. 태스크를 할당해도 사용률이 어느 한계를 넘지 않아, 현재의 동작 주파수를 유지할 수 있는 코어를 선택하거나 에너지 효율적인 코어에서 고성능 코어로 필요이상으로 스위칭하는 현상을 막는다면 소비전력 측면에서 최적화를 달성할 수 있다.

불행히도 비대칭 멀티코어 아키텍처의 코어 타입 선택 방식에서, 현재의 Linux CFS는 태스크를 할당할 때, 코어의 타입을 구분하거나 현재 코어가 얼마나 사용되고 있는지 고려하지 않는다. 이로 인하여 불필요한 동작 주파수 상승을 야기할 수 있다. 또한, 에너지 효율적인 코어에서 고성능 코어로 스위칭 동작을 과도하게 발생시킬 수 있다. 결과적으로, 이는 시스템 전체 소비전력을 필요이상으로 증가시키는 문제점을 야기시킨다. 이러한 문제점을 해결하기 위하여, Linux CFS의 태스크 할당정책에 현재 동작중인 코어의 사용 정도를 정량화하여

반영해야 한다. 이는 Linux CFS가 코어 타입 선택 방식을 위하여 에너지 효율적인 스케줄링 방식으로 개선되었다는 뜻이 된다.

1.2 전체 코어 사용 방식에서 공정할당 스케줄링 최적화

전통적인 임베디드 시스템에서 사용되는 멀티코어 스케줄링은 빠른 사용자 반응성 혹은 향상된 단위시간당 데이터 처리량에 집중되었다. 최근에 공정할당(fair-share) 스케줄링은 실제 활용 사례와 여러 가지 많은 연구결과들에서 중요한 기술로 다루어졌다. 공정할당 스케줄링은 태스크들에게 부여된 가중치(weight)에 비례하여 태스크들이 코어를 사용하게 하는 방식이다. 이러한 스케줄링 정책은 운영체제 수준에서 QoS(Quality-of-Service)를 제공한다 [2][3]. 현재의 공정할당 스케줄링은 대부분이 대칭 멀티코어 아키텍처를 위하여 개발되었고, 비대칭 멀티코어 아키텍처용 공정할당 스케줄링 알고리즘은 아직 개발이 미흡하여, 늘어나는 추세의 비대칭 멀티코어 아키텍처에 적절히 대응하지 못하고 있다.

앞서 설명한 비대칭 멀티코어가 주는 아키텍처적 이로운 점에도 불구하고, 효과적인 공정할당 스케줄링 정책을 개발하는 것은 이러한 아키텍처가 본연적으로 가지고 있는 코어간 성능 비대칭성 때문에 쉽지 않은 일이다. 대칭 멀티코어 아키텍처 환경에서는, 각 코어의 처리 능력이 동일하다. 그래서 공정할당 스케줄러는 실행 가능한 태스크들에게 CPU 시간을 할당할 때, 단순히 각 태스크들의 가중치에 비례하게 부여하면 된다. 이와 대비하여 비대칭 멀티코어 아키텍처에서는 고성능 코어와 에너지 효율적인 코어간의 처리 능력이 많이 다르다. 이러한 이유로 한 태스크에게 주어지는 CPU 시간은 태스크가 사용하는 코어의

처리 능력에 따라서 스케일(scale) 되어야 한다.

또한, CFS는 태스크들의 virtual runtime을 동일하게 유지하려고 노력한다. 하지만 이는 개별 코어에 국한되고, 시스템 전체적으로 태스크간 virtual runtime은 비슷하게 유지 되지 않는다. 이는 멀티코어간 부하분산을 virtual runtime을 기반으로 하지 않고, 태스크들의 가중치에 기반하여 부하를 분산하기 때문이다.

비대칭 멀티코어 아키텍처의 전체 코어 사용 방식에서, Linux CFS는 태스크들의 CPU 시간을 계산할 때, 코어의 처리 능력을 고려하지 않고 대칭 멀티코어 아키텍처와 동일한 방식으로 계산한다. 이에, 코어의 처리 능력을 고려한 스케일된 CPU 시간을 Linux CFS에 반영하고, 시스템 전체적으로 태스크간 virtual runtime을 비슷하게 유지한다면, Linux CFS가 비대칭 멀티코어 아키텍처를 위한 공정할당 스케줄링을 수행한다는 뜻이 된다.

제 2 절 논문의 기여

본 학위논문에서는 비대칭 멀티코어 아키텍처의 코어 타입 선택 방식과 전체 코어 사용 방식에 사용되는 최적화된 스케줄링 기법 연구에 대한 내용을 기술한다. 구체적으로, 코어 타입 선택 방식에서는 저전력 스케줄링 최적화를, 전체 코어 사용 방식에서는 코어간의 성능 비대칭성을 스케줄러에 반영하고, 최적화된 공정할당 스케줄링을 수행한다.

본 학위논문의 학술적, 기술적 기여를 요약하면 다음과 같이 나타낼 수 있다.

- **비대칭 멀티코어 아키텍처용 CFS 분석**

본 연구에서는 비대칭 멀티코어 아키텍처를 지원하기 위한 Linux CFS를 분석하고, 코어 타입 선택 방식과 전체 코어 사용 방식에서 각각의 모드를 위해서 어떤 기능들을 추가했는지 정리하였다. 분석 결과, 코어 타입 선택 방식에서는 코어내부에서의 스케줄링과 코어간 부하분산 정책은 기존의 Linux CFS를 유지하고, CPU 드라이버 소프트웨어를 통하여 고성능 코어와 에너지 효율적인 코어간 스위칭을 수행함을 알았다.

전체 코어 사용 방식에서는 코어 내부의 스케줄링은 기존 CFS를 유지하되, 코어간 부하분산 정책은 기존의 CFS의 부하분산정책을 같은 코어 타입간의 부하분산 정책에 적용하였다. 서로 다른 코어 집합간의 부하분산 정책은 별도의 태스크 이주 정책을 사용하고 있음을 알았다. 또한, 이러한 부하분산 정책들로 인하여, 공정할당 측면에서의 최적화되지 못한 점을 발견하였다.

- **코어 타입 선택 방식으로 동작 시 저전력 스케줄링 측면에서의 문제점 파악**

본 연구에서, 현재의 Linux CFS가 코어 타입 선택 방식에서 스케줄링을 수행할 때, 코어의 타입이나 주파수를 고려하지 않고 태스크들의 가중치만 고려하여 부하분산을 수행함을 찾았다. 이러한 부하분산 정책은 동작 주파수를 불필요하게 상승 시킬 수 있다. 또한, 에너지 효율적인 코어에서 고성능 코어로 스위칭 횟수를 증가시킬 수 있다. 이러한 점들은 전체적인 에너지 소비를 증가시킨다.

- **코어 타입 선택 방식에서의 DVFS 모델**

코어 타입 선택 방식에서 저전력을 고려한 부하분산 정책의 목표를 달성하기 위하여 본 연구에서는 먼저, 코어의 주파수 변동과 고성능 코어와 에너지 효율적인 코어간 스위칭을 발생시키는 메커니즘을 분석하였다. 이

를 위하여 Linux ONDEMAND governor의 DVFS 모델을 상태도(State Diagram)에 나타내었다 [4]. 코어 사용률과 주파수 크기를 각각 3단계 및 2단계로 정의하고, 이를 (주파수, 사용률) 형태로 나타내어 코어의 상태를 표현하였다. 코어의 상태변화를 나타냄으로써 단순화된 DVFS 모델을 얻을 수 있었다.

- **코어 타입 선택 방식에서의 사용률인지 기반 부하분산 (utilization-aware load balancing) 알고리즘**

상태도의 DVFS 모델을 통해서, 코어의 동작 주파수 변경과 코어 타입 간 스위칭의 정확한 메커니즘을 분석할 수 있었다. 분석된 결과를 사용하여, 본 연구에서는 코어 타입 선택 방식에 적합한 사용률인지 기반 부하분산 알고리즘(utilization-aware load balancing)을 제안하였다. 이 알고리즘은 각 코어의 실행 큐(run-queue)에 있는 태스크들을 분산시킨다. 어떤 태스크가 시스템의 준비 큐(wait-queue)에서 빠져 나와 할당될 실행 큐를 찾을 때, 가장 사용률이 낮은 코어에 할당하게 함으로써 코어의 주파수 상승이나 에너지 효율적인 코어에서 고성능 코어로의 스위칭 발생 빈도를 최대한 줄인다. 실험 결과를 토대로, 기존 Linux CFS 대비 최대 11.35% 소비 전력 감소를 나타냄을 보인다.

- **코어 타입 선택 방식에서의 사용률 기반 추정기(utilization-based estimator)**

사용률인지 기반 부하분산 알고리즘이 준비 큐에서 빠져 나온 태스크를 실행 큐에 할당할 때, 이 태스크로 인한 해당 코어의 사용률 증가를 예측할 필요가 있다. 이를 위하여 본 연구에서는 코어의 사용률기반 추정기(utilization-based estimator)를 제안한다. 이 추정기를 통하여 사용률 증가가 가장 작은 코어를 알아내고, 태스크를 그 코어의 실행 큐에 할당한다.

■ **전체 코어 사용 방식으로 동작 시 공정할당 스케줄링 측면에서의 문제점 파악**

첫째, 전체 코어 사용 방식으로 동작 시 Linux CFS는 코어간 성능 비대칭성을 고려하지 않는 CPU 시간을 사용하며, 이를 CFS의 virtual runtime 산정 시 그대로 사용함을 알았다. 이는 태스크간의 CPU 공정할당(fair-share)성을 훼손한다는 것을 밝혔다.

둘째, CFS는 태스크들의 virtual runtime을 동일하게 유지하려고 노력한다. 이는 태스크들의 상대적인 진척도를 비슷하게 유지시키기 위함이다. 하지만 이러한 노력은 개별적인 코어에서는 유지되나, 시스템 전체적으로 유지 되지 않는다. 이러한 이유로, 시스템의 수행 시간이 늘어남에 따라서 태스크간 virtual runtime 차이는 증가된다. 이는 태스크간 상대적인 수행 정도 차이가 더욱 더 커지게 하는 문제점을 발생시킨다.

■ **전체 코어 사용 방식을 위한 SVR(Scaled Virtual Runtime) 정의**

Linux CFS는 virtual runtime을 사용하여 태스크의 상대적 진척 정도를 나타낸다 [39]. 이는 태스크의 가중치에 반비례하고, CPU 시간에 비례한다. 이 때, CPU 시간은 대칭 멀티코어 아키텍처임을 가정하고 산정한다. 본 연구에서는 스케일된 CPU 시간을 제안한다. 이는 코어간 성능 비대칭성을 고려한 CPU 시간이다. 또한 같은 코어 타입에서도 주파수에 따라서 나타내는 성능이 다르다. 따라서 주파수에 따른 성능 차이도 스케일된 CPU 시간에 반영하였다. 이렇게 구해진 스케일된 CPU 시간을 Linux CFS의 virtual runtime 계산에 적용하였다. 이를 스케일된 virtual runtime 즉, SVR(scaled virtual runtime)로 정의하였다.

■ **전체 코어 사용 방식을 위한 SVR 기반 공정할당 스케줄링 (SVR-based fair-share scheduling)**

비대칭 멀티코어의 공정할당 문제를 해결하기 위해서, 본 학위논문에서는 SVR 기반 공정할당 스케줄링(SVR-based fair-share scheduling) 알고리즘을 제안한다. 본 연구는 태스크들의 상대적인 진척 정도를 SVR(scaled virtual runtime)로 정의하고, 이를 부하분산을 통해서 균등하게 맞춘다. 비대칭 멀티코어 아키텍처는 고성능 코어집합인 고성능 클러스터와 에너지 효율적인 코어집합인 클러스터로 이루어지며, 본 연구에서는 각각의 클러스터 내부에서 태스크들의 진척 정도 차이를 작은 상수로 제한시킨다. 이로써 공정할당 스케줄링의 최적화 목표를 달성한다.

수학적 분석을 통하여 임의의 두 개의 태스크간 SVR 차이 값은 작은 상수 이내로 제한된다는 것을 밝혔다. 공정할당 스케줄링의 보다 직관적인 효과를 보기 위하여 여러 개의 같은 태스크들을 실행시키고 그들의 완료시간을 측정하였다. 그 결과, 기존 방식 대비, 태스크들의 완료시간 표준 편차가 56 %감소됨을 보였다.

제 3 절 논문 구성

본 학위논문의 나머지 부분의 구성은 다음과 같다. 2장에서는 관련 연구 및 배경 지식을 설명한다. 3장에서는 비대칭 멀티코어 아키텍처의 코어 타입 선택 방식에 최적화된 저전력 스케줄링 기법을 4장에서는 비대칭 멀티코어 아키텍처용 전체 코어 사용 방식에 최적화된 공정할당 스케줄링 기법을 설명한다. 5장은 본 연구에서 제안한 두 가지 스케줄링 알고리즘에 대한 실험적 검증을 기술하고, 아울러 수학적 분석을 통한 제안된 알고리즘의 실효성을 분석한다. 마지막 6장은 본 학위논문의 결론부이며, 향후 연구 방향에 대하여 기술한다.

제 2 장　　관련 연구 및 배경 지식

본 장에서는 본 학위논문의 토대가 되는 기존에 수행된 관련 연구들에 대해서 정리하여 기술한다. 또한, 본 연구를 수행함에 있어서 관련된 배경 지식을 소개한다. 구체적으로, 멀티코어 아키텍처의 스케줄링 관련 연구들 중 저전력 운용기법관련 기존 연구를 소개한다. 이어, 공정할당 스케줄링 관련 연구들을 기술한다. 최근 비대칭 멀티코어 아키텍처의 대표적인 예는 ARM사의 빅리틀 아키텍처이다. 따라서 본 연구에서는 이 아키텍처를 실험 대상으로 하였다. 본 장에서 다루는 배경 지식은 이의 하드웨어적, 소프트웨어적 특징들을 설명한다.

제 1 절　관련연구

1.1 멀티코어 아키텍처용 저전력 스케줄링

모든 코어들이 같은 명령어 집합(Single-ISA)을 사용하는 비대칭 멀티코어 아키텍처는 많은 연구들의 대상이 되어 왔다 [1][28]. 특히, 그들 중 대부분의 연구들은 이러한 아키텍처상의 프로세서들에게 최적화된 방법으로 태스크들을 할당하는 연구에 중심을 두었다 [30][31]. 또한 많은 연구들이 에너지 효율성 보다는 최대한의 컴퓨팅 성능을 낼 수 있는 스케줄링 기법에 대한 연구를 수행해 왔다 [32][33]. 하지만, 최근 들어, 에너지 효율성에 대한 관심이 연구 주제로 채택되기 시작했다.

Cochran은 최적화된 DVFS 운용과 태스크할당을 위한 제어 기법을 제안했다 [34]. 이 연구에서는 온도, 소비전력 데이터, 성능 측정용 카운터 등을 이용하여 제한된 전력 예산 하에 최대 성능을 내기 위한 방법을 제안하였다. Pack & Cap이라 불리는 이 방법은 태스크들을 패킹(Packing)하여 코어들에 할당하고, 일부 유휴(idle) 코어들은 Sleep상태에 진입하게 하여, 소비전력 상한선(Capping)내에서 성능 손실 없이 동작하게 한다.

SmartCap은 자가적응(self-adaptive)형 소비전력 제어 기법을 제안한다 [35]. 이 기법은 사용자 체험에 기반한 샘플링된 통계 데이터를 사용한다. 이를 사용하여 멀티코어 아키텍처가 가장 작게 사용할 수 있는 주파수를 응용 프로그램 별로 구한 후 이를 Linux의 DVFS [36]에 반영하였다. 그 결과 주파수를 필요 이상으로 상향 조정(over-provisioning)하는 현상을 막을 수 있다.

HPM(Hierarchical Power Management)은 빅리틀 아키텍처에서 소비전력과 온도를 성능과 관련시켜 제어하는 기법을 제안하고 있다 [40]. 이 기법은 태스크들의 QoS를 고려하여 코어의 주파수를 throttling시키는 방법을 사용한다. 이를 위하여, HPM은 PID(proportional integral derivative)기반 제어 기법을 사용하며, 허락된 소비전력과 칩 온도 하에서 QoS를 떨어뜨리지 않는 수준으로 코어의 주파수를 동작시킨다.

위에 열거된 [34][35][40]은 코어의 주파수를 직접 제어하여 소비전력 상승을 억제하는 방법을 사용하고 있다. 이와 달리 본 연구에서는 Linux kernel의 부하분산 기법을 개선하여 에너지 효율성을 높이는 방법을 사용한다. 본 연구에서의 주된 관심사는 현재 사용되는 Linux kernel의 부하분산 기법은 코어의 사용률을 고려하지 않는다는 점이다. 하지만 Linux kernel의 DVFS 모듈은 코어의 사용률에 따라서 동작 주파수나

코어 타입을 결정한다. 부하분산 기법과 DVFS 모듈간의 이러한 커뮤니케이션 부재는 비대칭 멀티코어 아키텍처의 최적화되지 못한 소비전력 결과를 초래한다.

1.2 멀티코어 아키텍처용 공정할당 스케줄링

공정할당을 위한 멀티코어 아키텍처용 스케줄링은 다양하고 폭 넓게 연구되어 왔다. 이 연구들의 대부분의 목적은 서버와 같은 시스템에서 사용자들에게 각각 구분된 성능을 보장하기 위함이다. 이들은 크게 중앙(centralized) 실행 큐 알고리즘 기반과 분산(distributed) 실행 큐 알고리즘 기반 스케줄링으로 분류될 수 있다. 중앙 실행 큐 알고리즘 방식은 시스템상에 한 개만 존재하는 실행 큐를 사용하기에 간단하고 직관적이다 [10][11][12][13][29].

분산 실행 큐 알고리즘 방식은 실행 큐를 차지하기 위한 태스크들의 경쟁을 피하기 위해서 코어마다 자신의 실행 큐를 갖게 한다. 특히 이러한 방식은 큰 규모의 시스템에서 더욱 효과적이다. 이 알고리즘은 각 코어 별로 다른 코어와 상관없이 스케줄링을 수행한다. 따라서 코어간 태스크 이동은 피할 수 없다. 만약 태스크 이동이 없다면, 각 코어들의 부하 크기 차이는 시간에 따라서 증가된다. 태스크 이동 정책에 따라서 분산 실행 큐 알고리즘 기반 스케줄링 기법은 가중치(weight) 기반 방식 [14][15][16] 과 속도(rate) 기반 방식 [7][17][18][19]으로 나뉘어 진다. 코어간 공정할당 스케줄링을 위해서 가중치 기반 방식은 부하(load)의 개념을 사용하고, 속도 기반 방식은 페이스(pace)의 개념을 사용한다. 코어의 부하는 간단히 말해서, 실행 가능한 태스크들의 가중치 합을 나타내고, 코어의 페이스는 지금까지 수행한 라운드(round)를 뜻한다. 단일 코

어 별 스케줄링은 가중치 기반과 속도 기반 방식 모두 태스크의 가중치를 사용한다.

대칭 멀티코어 시스템은 공정할당 스케줄링을 수행 시 단순히 CPU 시간만 고려하면 문제가 없다. 하지만 비대칭 멀티코어 시스템은 코어의 성능 비대칭성을 반드시 고려해야 한다. 비대칭 멀티코어 시스템의 공정할당 스케줄링 알고리즘은 (1) 정확하게 코어간 성능 비대칭성을 정량적으로 나타내는 방식과 (2) 나타내지 않는 방식으로 나눌 수 있다.

Li는 AMPS(Asymmetric Multiprocessor Scheduler)를 제안하였다 [20]. 이 연구에서 공정성(Fairness)은 모든 코어들의 스케일된 부하를 주기적으로 분산시킴으로써 얻어진다. 코어의 실행 큐에 존재하는 태스크 수를 코어의 컴퓨팅 능력으로 나눈 값을 스케일된 부하로 정의하였다. 이 때, 컴퓨팅 능력은 $F \times S$로 정의되며, F는 코어의 동작 주파수이고 S는 offline 벤치마크 테스트로 구해진 스케일링 값이다. 코어의 컴퓨팅 능력은 실행 중에 스케줄러에게 알려진다. AMPS는 몇 가지 단점이 있다. AMPS는 코어의 컴퓨팅 능력을 고정된 상수 값으로 표현하였다. 하지만 실제 운용 중에는 코어의 상태에 따라서 컴퓨팅 능력은 동적으로 다양하게 변한다. 더욱더, AMPS는 태스크들의 가중치를 고려하지 않아서 공정성(fairness) 본연의 의미를 지키지 못한다.

Li는 또한 A-DWRR(Asymmetric-Distributed Weight Round-Robin)을 제안하였다 [1]. 이는 DWRR의 확장된 버전이다. DWRR에서 제안한 태스크의 공정성은 round 즉, 속도의 개념으로 나타내었다 [7]. 한 round는 시스템에서 실행 가능한 모든 태스크들의 time slice들의 합으로 나타내었다. DWRR은 코어간 태스크 이동을 수행한다. 이러한 이동의 목적은 모든 태스크들의 round 값들이 같게 하기 위함이다. DWRR은 round 단위로 동작하기에 아주 정밀한 멀티코어에서의 공

정성을 보장하지는 않는다. 불행히도, A-DWRR은 이 특징을 그대로 가지고 있다. 코어간 비대칭성을 나타내기 위해서 A-DWRR은 상대적인 CPU 시간을 사용한다. 상대적인 CPU 시간은 CPU rating 값으로 스케일된 CPU 시간을 말한다. CPU rating이란 어떤 코어와 가장 느린 코어의 성능 차이를 숫자로 나타낸 값을 의미한다. 공정할당을 위하여 A-DWRR은 각 태스크들이 그들의 가중치에 비례하여 스케일된 CPU 시간을 갖도록 한다. A-DWRR은 코어의 CPU rating을 고정된 상수로 나타내었다. 따라서 DVFS에 따른 코어의 동작 주파수 변화는 CPU rating에 영향을 주지 못하는 단점이 있다.

Craeynest는 equal-progress 스케줄러를 제안하였다 [2]. 이 스케줄러는 공정할당을 위해서 모든 태스크들의 slowdown을 비슷하게 유지한다. 태스크의 slowdown은 실제 수행 시간과 고성능 코어 단독에서 수행했다고 가정한 시간의 비율을 나타낸다. 실제 환경에서 태스크는 고성능 코어와 에너지 효율적인 코어를 옮겨 다니며 수행된다. 따라서 어떤 태스크의 고성능 코어에서 단독으로 수행했다고 가정한 시간을 구하기 위해서는 에너지 효율적인 코어에서 수행한 시간들을 고성능 코어에서의 수행시간으로 변환해야 한다. 이를 위하여, 어떤 태스크가 수행될 때 사용한 두 가지 코어타입간의 CPI(cycle per instruction) 비율을 사용하여 변환한다. Equal-progress 스케줄러는 slowdown 값이 큰 태스크들을 고성능 코어에 할당하면서 공정할당 스케줄링을 수행한다. 하지만 이 스케줄러 역시 AMPS와 마찬가지로 태스크들의 가중치를 고려하지 않았다 [20].

[1][2][20]과 달리 비대칭성을 어떤 값으로 표현하지 않은 스케줄링 방식은 묵시적인 방법으로 성능 비대칭성을 고려하였다. Kazempour는 AASH(Asymmetry Aware Scheduler for Hypervisors)를 제안하였다

[21]. AASH는 hypervisor에서 수행되는 태스크들에 대한 공정할당 스케줄링이다. 이 스케줄러는 모든 테스크들이 고성능 코어에서 수행한 cycle 수와 에너지 효율적인 코어에서 수행한 cycle 수를 같게 유지한다. 이를 위하여 Xen hypervisor의 credit 스케줄러를 수정하였다. AASH는 주기적으로 어떤 태스크들이 고성능 코어에서의 cycle 수가 지나치게 높은지 체크한 후, 이들을 에너지 효율적인 코어로 이동시킨다.

R%-fair 스케줄러는 공정할당과 시스템의 처리량(throughput)을 동시에 고려한 스케줄러이다 [22]. 이 스케줄러는 고정된 분량(R%)의 고성능 코어에서의 cycle 수를 모든 태스크들이 균등하게 갖도록 한다. 나머지 분량((100-R)%)의 고성능 코어 cycle 수는 높은 처리량(throughput)을 필요로 하는 태스크들에게 할당된다. AASH와 마찬가지로 Xen hypervisor의 credit 스케줄러를 수정하였다.

AASH, R%-fair [21][22] 두 스케줄러는 시스템의 모든 태스크들이 같은 고성능 코어 cycle 수와 에너지 효율적인 코어 cycle 수를 갖도록 만드는 기법을 사용하였다. 이들은 본 연구에서 제안하는 스케줄링 알고리즘과 다른 기법이며, CPU 시간 할당에 있어서 태스크들의 가중치를 사용하지 않은 점을 생각하면 본연의 공정할당의 의미와는 거리가 먼 스케줄링 기법이다.

제 2 절 배경 지식

본 절에서는 본 학위논문의 나머지 부분에 대한 이해를 돕기 위하여 대상 시스템에 대한 설명을 한다. 본 학위논문에서는 대상 시스템으로, 비대칭 멀티코어 아키텍처 중 대표적으로 많이 쓰이는 ARM사의 빅리틀 멀티코어 아키텍처를 선정하였다. 이 아키텍처는 스마트폰, 테블릿, DTV

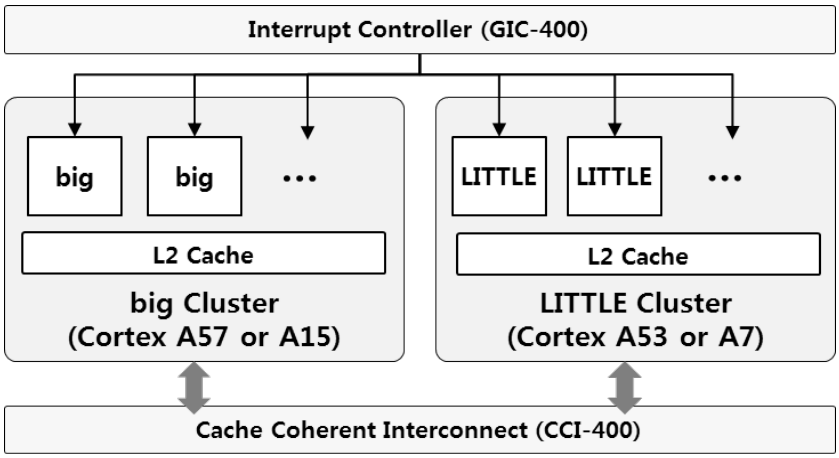


그림 2. 빅리틀 아키텍처

등의 상용 임베디드 시스템에서 저전력, 고성능을 목적으로 가장 많이 채택되는 아키텍처이다.

빅리틀 아키텍처는 두 개의 서로 다른 타입의 코어들로 이루어져 있다. 32 비트 시스템을 위하여 Cortex-A15는 빅코어, Cortex-A7은 리틀코어로 사용된다. 64 비트 시스템에서는 Cortex-A57과 Cortex-A53이 각각 빅코어와 리틀코어로 사용된다. 같은 타입의 코어들은 한 개의 클러스터에 속하게 된다. 이때, 빅코어들과 리틀코어들이 속한 클러스터는 각각 빅클러스터와 리틀클러스터이다. 그림 2는 클러스터 구조에 기반한 ARM사의 전형적인 빅리틀 아키텍처를 보여준다. 그림에서 CCI-400은 각 클러스터간의 공유된 cache 데이터들의 일관성을 보장하기 위하여 사용되는 하드웨어 장치이다. 두 개의 클러스터 혹은 각각의 빅리틀 코어는 GIC-400을 통해서 인터럽트 요청을 받게 된다 [5][6][27][37].

빅리틀 아키텍처에서 사용되는 비대칭 멀티코어 아키텍처의 코어 타입 선택 방식과 전체 코어 사용 방식은 각각 Switcher 모드와 GTS(Global

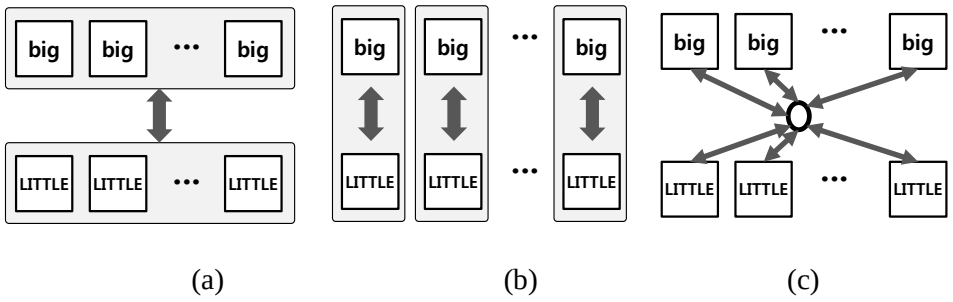


그림 3. 빅리틀 아키텍처의 동작 모드

Task Scheduling) 모드이다 [27]. 그림 3은 이러한 모드들을 보여주고 있다. 먼저 Switcher는 빅과 리틀의 두 가지 코어타입에서 어떤 코어를 사용할 지를 결정하는 Linux kernel의 소프트웨어 모듈이다 [38]. Switcher 모드는 클러스터간 이주 모드와 CPU간 이주 모드 두 가지 모드로 나누어지며, 그림 3의 (a)와 (b)가 각각 나타내고 있다. 클러스터간 이주 모드에서는 빅리틀 아키텍처가 실행 중에 빅클러스터와 리틀클러스터 중 한 개의 클러스터만 동작시키게 한다. CPU간 이주 모드에서는 한 개의 빅코어와 한 개의 리틀코어가 한 쌍을 이루며 이를 pair라고 부른다. 이 모드에서는, 실행 중에 하나의 pair에 속한 빅코어와 리틀코어 중 한 개의 코어만 동작시킨다.

그림 3의 (c)는 GTS 모드를 나타낸다. 이 모드에서는 모든 빅코어와 리틀코어는 각각 독립적으로 on/off 되며, 모든 코어들은 언제나 사용 가능하다. 따라서, 빅리틀 아키텍처의 GTS 모드는 그림 3에 나타난 세가지 운용모드 중 가장 좋은 성능과 스케줄링에 있어서 가장 좋은 유연성을 나타낸다.

이러한 하드웨어적 운용모드를 이용하여, 저전력이 요구되는 시스템에서는 Switcher 모드를 사용하고, 고성능이 필요한 시스템에서는 GTS

모드를 사용하도록 ARM사의 빅리틀 아키텍처는 지원하고 있다. 또한, Linaro 스케줄링 프레임웍은 ARM사의 빅리틀 아키텍처의 Switcher 모드와 GTS 모드에 필요한 각각의 소프트웨어 프레임웍을 제공하고 있다 [41]. 본 연구에서는 32 비트 아키텍처를 사용하여, 빅코어는 Cortex-A15를 나타내고, 리틀코어는 Cortex-A7을 사용하였다.

제 3 장 비대칭 멀티코어 아키텍처용 저전력 스케줄링

본 장에서는 비대칭 멀티코어 아키텍처에서 사용되는 최적화된 저전력 스케줄링 기법에 대하여 기술한다. 이를 위하여, 코어 타입 선택 방식인 Switcher 모드를 지원하는 ARM사의 빅리틀 아키텍처를 대상 시스템으로 정의한다. 그림 4는 빅코어인 Cortex-A15와 리틀코어인 Cortex-A7의 성능과 소비전력의 관계를 나타내고 있다. Switcher 모드로 동작 시, 그림에 나타낸 소비전력과 성능의 관계를 고려해서 코어의 상태를 빅코어 혹은 리틀코어로 제어한다.

Switcher 모드는 앞 장에서 설명한 바와 같이 클러스터간 이주 모드와 CPU간 이주 모드로 나누어진다. 클러스터간 이주 모드에서는 실행

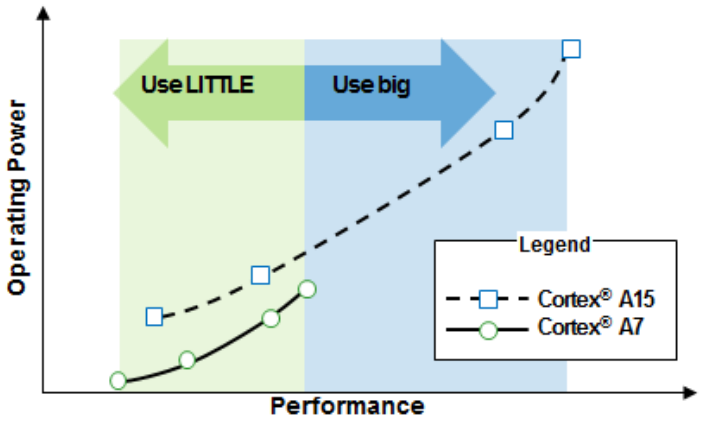


그림 4. 빅리틀 아키텍처의 성능과 소비전력의 관계

중에 항상 같은 타입의 코어들만 운용된다. 우선권은 항상 빅코어에 주어진다. 따라서, 시스템이 리틀코어로만 동작하는 중에, 리틀코어들 중 한 개의 코어만 빅코어를 필요로 한다고 하더라도, 모든 코어가 빅코어로 스위칭된다 [5][6].

클러스터간 이주 모드와 달리 CPU간 이주 모드에서는 각각의 빅리틀 pair 중 한 개의 코어가 사용되며, 다른 pair들의 상태와는 무관하게 동작한다. CPU간 이주 모드에서는 각 pair의 빅코어가 필요 없을 때는 off 시킨 후 리틀코어를 사용하므로 빅리틀 아키텍처에서의 에너지 효율성 극대화를 기대할 수 있다. 따라서 본 연구에서는 Switcher 모드 중 CPU간 이주 모드만 대상으로 하고, 이러한 하드웨어적인 특성에 최적화된 저전력 스케줄링 알고리즘을 제안한다.

제 1 절 시스템 정의

본 절에서는 빅리틀 아키텍처의 Switcher모드 중, 본 학위논문에서 대상으로 하는 CPU간 이주 모드가 사용하는 하드웨어적, 소프트웨어적 요소들에 대하여 설명한다.

1.1 가상 CPU와 부하분산

CPU간 이주 모드에서, 각 빅리틀 pair는 한 개의 가상 CPU로 나타내어진다 [38]. Linaro 스케줄링 프레임웍은 Linux CFS [39]를 사용하여 태스크들을 가상 CPU들에 할당하면서 스케줄링을 수행한다. 본 연구에서는 이러한 가상 CPU들의 집합을 S 라 정의하고 다음과 같이 나타낸다.

$$S = \{vCPU_0, vCPU_1, vCPU_2, vCPU_3\} \quad (1)$$

그림 5는 이에 대한 동작 예시를 나타낸다. 그림의 빗금 친 부분은 power-off된 상태의 코어를 나타낸다. 그림에서 알 수 있듯이 pair로 이루어진 가상 CPU 중 한 개의 코어만 동작하며, 각각의 가상 CPU들은 다른 가상 CPU의 상태에 상관없이 자신의 코어 타입을 결정한다.

Linux CFS는 집합 S에 속해있는 가상 CPU들 사이에서 태스크들을 옮긴다. 이때, 가상 CPU들 사이에서 직접 옮기기도 하고, 때로는 시스템의 준비 큐(wait-queue)에 태스크를 넣었다가 다시 꺼내어 가상 CPU들의 실행 큐(run-queue)에 할당하기도 한다. Linux CFS는 이러한 부하분산을 수행 시에 가상 CPU의 물리적 상태 즉, 빅코어인지 리틀코어인지, 혹은 가상 CPU의 동작주파수는 얼마인지 상관하지 않는다.

시스템의 준비 큐는 IO 동작으로 인하여 블로킹되거나 Sleep 상태로 진입한 태스크들을 가지고 있다. 실행 큐 q_s는 가상 CPU $s \in S$에서 수행할 태스크들의 집합을 나타낸다. Linux CFS는 가상 CPU들의 부하(load)를 고려하여 태스크들을 할당하고, 각 가상 CPU들간의 부하를 동일하게

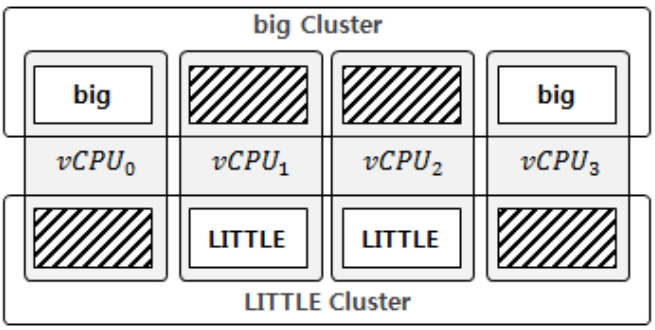


그림 5. CPU간 이주 모드 예시

맞추려고 노력한다. 이때 가상 CPU $s \in S$의 부하는 다음과 같이 정의된다.

$$L_s = \frac{\sum_{\tau \in T_k} w(\tau)}{\Gamma_s} \quad (2)$$

T_s는 q_s에 속해있는 실행 가능한 전체 태스크 집합을 나타내며, $w(\tau)$는 태스크 τ의 가중치를 나타낸다. Γ_s는 q_s가 할당된 가상 CPU s의 컴퓨팅 능력을 나타내는 상수이다. 이는 시스템의 아키텍처 특성에 따라서 미리 정해진 값이다. 이러한 방법으로 실행 큐에 할당되는 부하는 그 실행 큐가 할당된 가상 CPU의 현재 컴퓨팅 능력에 비례하여 할당된다.

실행 큐간의 적절한 부하분산을 위하여 두 가지 서로 다른 부하분산 정책이 사용된다. 첫째, 주기적 부하분산 정책이 있으며, 둘째, 유휴상태 부하분산 정책이 있다. 주기적 부하분산 정책은 주기적으로 각각의 실행 큐들의 부하를 살피고 그 차이가 어떤 불균형 상한 값을 넘는지를 체크한다. 이 때, 불균형 상한 값은 가장 큰 부하 값에서 현재 가상 CPU의 실행 큐의 부하 값을 뺌으로써 구해진다. 또한 이때 옮겨지는 태스크의 양은 불균형 정도에 따라 정해진다.

어떤 가상 CPU의 현재 수행되는 태스크가 할당 받은 time slice를 다 소진할 때까지 실행 큐에 다른 태스크들이 존재하지 않는다면 Linux CFS는 유휴상태 부하분산을 수행한다. 이 시점에서, 가장 부하 값이 큰 실행 큐로부터, 태스크들을 비어있는 실행 큐로 옮긴다.

1.2 Governor와 코어의 사용률

어떤 가상 CPU의 동작 주파수와 사용률은 현재 그 코어 위에서 수행되는 태스크들에 따라서 정해진다. Linux의 DVFS 정책은 코어의 사용률에 따라서 주파수를 변화시킨다. 높은 성능이 요구될 때는 코어의 사용률이 높아지며, 이에 반응하여 DVFS 정책은 governor를 통해서 코어의 주파수를 상승시킨다. 반대의 경우 소비전력을 최소화하기 위해서 코어의 주파수를 최소로 줄인다. 가상 CPU $s \in S$의 사용률은 다음과 같이 정의된다.

$$U(s) = 100 \times \frac{(Period - Idle\ time)}{Period} \tag{3}$$

위 식에서 $Period$ 는 연속된 governor epoch의 시간 축 상의 간격을 나타내며, 이 때 governor epoch는 ONDEMAND governor가 코어의 사

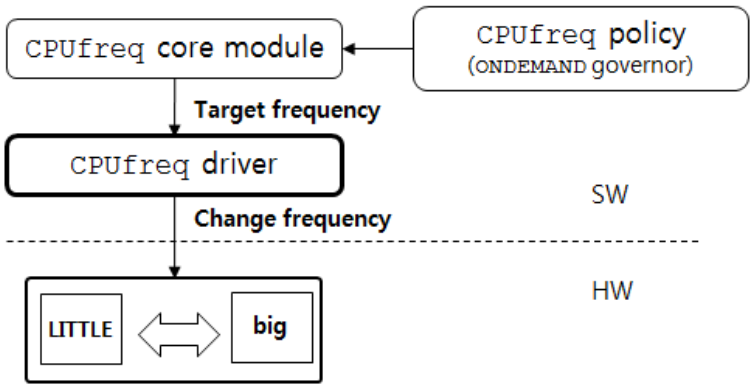


그림 6. Governor의 주파수 및 코어 타입 변경 동작 예시

용률을 관찰하기 시작한 시간이다 [36]. ONDEMAND governor는 Linux kernel에 구현되어 있다. 이는 코어의 사용률에 따라서 주파수를 조절하는 모듈이다.

그림 6은 governor가 실제 하드웨어를 제어해서 주파수를 변화시키거나 코어의 타입을 변경시키는 동작을 설명하고 있다. 그림에서 알 수 있듯이, governor에 의하여 정책이 결정되면 CPUfreq 모듈에 그 시그널이 입력된다. CPUfreq 모듈은 DVFS 정책에 정의된 주파수 값을 참고하여 목표로 하는 주파수 값을 결정한다. 이 결정된 값은 드라이버에 전달되고, 드라이버는 직접 코어를 제어하여 주파수를 변경시킨다. 이때 전달되는 주파수 값과 코어의 현재 상태에 의하여 코어 타입이 변화되며, 이에 대한 설명은 다음 세부 절에 나타낸 상태도를 통하여 자세히 설명한다.

ONDEMAND governor 모듈은 코어의 사용률을 주기적으로 관찰한다. 이 때 사용률의 상한 값과 하한 값을 사용한다. Governor는 한 관찰 구간 사이에서 관찰된 코어의 사용률이 상한 값(80%)를 넘을 경우, 가상 CPU가 취할 수 있는 가장 높은 주파수 값으로 현재의 주파수를 변경한다. 반대로 하한 값(70%)보다 낮을 경우 현재의 주파수에서 한 단계 낮은 주파수로 변경한다.

1.3 CPU간 이주 모드에서의 DVFS 모델

빅리틀 아키텍처에 사용되는 ONDEMAND governor 동작을 나타내기 위해서, 우리는 단순화된 DVFS 모델을 사용한다. 이모델을 위하여, 코어의 동작 주파수는 두 가지 상태로 정의한다. {low, high}의 두 가지 동작 주파수를 나타내는 상태는 줄여서 {L, H}로 표시한다. 다음으로, 코어의 사용률은 세가지 상태로 정의한다. {low, medium, high}의 세가지 사용

률을 나타내는 상태는 줄여서 {L, M, H}로 표시한다. 따라서 Cortex-A15/Cortex-A7로 이루어진 pair는 6가지 상태 중 한 개로 표현된다. 이는 (주파수, 사용률)로 표기되며 모두 열거하면 다음과 같다: (L, L), (L, M), (L, H), (H, L), (H, M), (H, H). 그림 7은 이들의 상태 변이를 나타낸 상태도 이다. 가로 축은 {L, H} 값으로 표현되는 코어의 동작 주파수를 나타내고, 세로 축은 {L, M, H} 값으로 표현되는 코어의 사용률을 나타낸다.

실제 운용 중에는, 코어가 취할 수 있는 많은 종류의 주파수가 존재한다. 또한, % 값으로 측정되는 다양한 값의 사용률이 있다. 이러한 주파수와 사용률 값을 간단하게 {L, H}와 {L, M, H}로 표현하고자, 표 1과 표 2에 간략히 정리된 이들의 정의를 나타내었다.

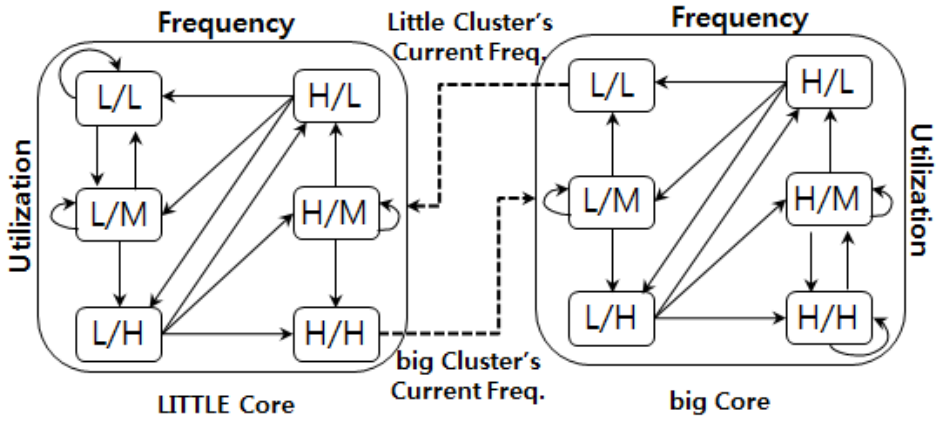


그림 7. 가상 CPU의 상태도

표 1. 가상 CPU의 주파수 상태의 의미

상태	L	H
주파수(f)	f < 최대 허용 주파수	f = 최대 허용 주파수

표 2. 가상 CPU의 사용률 상태의 의미

상태	L	M	H
사용률(U)	U<하한 값	하한값 <U <상한값	U>상한 값

그림 7에서 보여주듯이, 리틀코어가 (H, H) 상태에 있을 때만 다음 상태에서 빅코어로 스위칭되는 것을 알 수 있다. 새롭게 활성화된 빅코어의 주파수는 빅클러스터의 현재 주파수로 결정된다. 그리고 새롭게 활성화된 빅코어의 사용률은 빅클러스터의 주파수와 할당된 태스크들에 따라서 결정된다.

반대로, 빅코어가 (L, L) 상태에 있을 때만 다음 상태에서 리틀코어로 스위칭된다. 그리고 이때, 빅코어의 실행 큐에 있던 태스크들은 새롭게 활성화된 리틀코어에 할당된다. 이는 pair로 이루어진 가상 CPU는 한 개의 실행 큐가 있고, 같은 pair 내부의 빅코어와 리틀코어는 이 실행 큐를 공유하기 때문이다. 새롭게 활성화된 리틀코어의 주파수는 현재 리틀 클러스터에서 사용되는 주파수로 설정되며, 이 주파수와 할당된 태스크들에 따라서 사용률이 결정된다.

그림 8은 가상 CPU $s \in S$가 시간에 따라서 상태가 변경되는 모습을 예를 들어서 설명하고 있다. 각 governor epoch들은 점선으로 이루어진 화살표로 나타내었다. 그리고 이들의 시간 축 상의 표기는$GE_{(s,x)}$로 하였

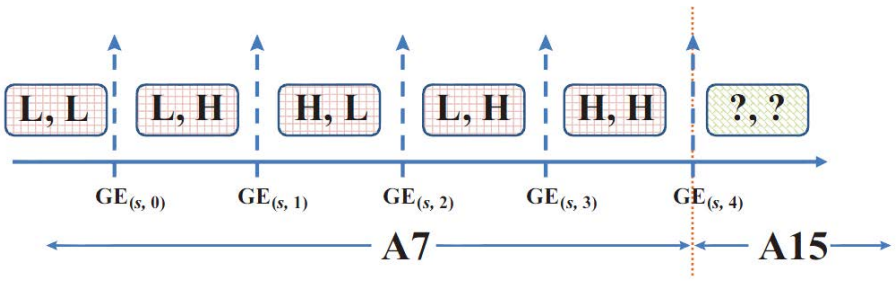


그림 8. 가상 CPU의 상태 변화 예시

으며, x는 governor epoch의 순서를 나타낸다. ONDEMAND governor는 각 governor epoch에서, 현 시점과 바로 직전의 governor epoch 사이의 코어의 사용률을 측정한다. 측정된 값에 따라서, governor는 가상 CPU의 주파수를 조절하거나 pair 안에서 코어간 스위칭을 수행한다.

그림 8의 예를 들어 설명하면 다음과 같다. 가상 CPU의 상태가 리틀 코어에서 시작한다면, $GE_{(s,1)}$에서 governor는 $GE_{(s,0)}$과 $GE_{(s,1)}$사이에서 코어 사용률이 'H'임을 착안하고 $GE_{(s,1)}$과 $GE_{(s,2)}$사이의 코어의 동작 주파수를 'H'상태로 변경한다. $GE_{(s,2)}$에서는 $GE_{(s,2)}$과 $GE_{(s,1)}$사이에서 코어의 사용률이 'L' 상태임을 착안하고 동작 주파수를 한 스텝 낮은 주파수로 설정한다. 따라서 주파수는 'L' 상태로 변경된다. 코어의 사용률은 $GE_{(s,2)}$과 $GE_{(s,3)}$사이에서 다시 높아짐에 따라서 governor는 동작 주파수를 상승시킨다. 마침내 $GE_{(s,3)}$과 $GE_{(s,4)}$사이에서 주파수와 사용률이 (H, H) 상태가 되고, $GE_{(s,4)}$에서 빅코어로 스위칭된다. 코어의 상태 즉 (주파수, 사용률)은 실행 중에 결정된다. 따라서 $GE_{(s,4)}$이후 상태는 (?, ?)로 표시하였다. 할당된 주파수는 빅클러스터의 주파수로 할당된다. 이때 모든 빅코어가 off 상태였다면, DVFS 정책에 의하여 빅코어의 가장 낮은 주파수로 설정된다

제 2 절 문제 정의

Linaro 스케줄링 프레임웍에서 사용하는 Linux CFS는 코어의 사용률을 고려하여 부하분산을 수행하지 않는다. 이로 인하여, 어떤 태스크가 필요 없이 높은 주파수로 동작하는 코어에 할당하거나 혹은 필요 없이 어떤 코어의 주파수를 상승시킬 수 있다. 또한, 불필요한 리틀코어에서 빅코어로 스위칭을 야기할 수 있다. 이러한 현상은 모두 코어들이 필요 이상의 에너지를 사용하도록 만드는 원인을 제공한다.

예를 들면, 두 개의 가상 CPU $s_1, s_2 \in S$가 있고, s_1, s_2 모두 현재 리틀 코어 상태이며 모두 마지막 governor epoch 시점에서 (L, L) 상태라고 하자. 그리고 다음의 식을 만족한다고 가정한다.

$$\mathrm{L}_{s_1} < \mathrm{L}_{s_2} ,$$

$$U(s_1) > U(s_2)$$

Linux CFS에게는, 시스템의 준비 큐에서 빠져 나온 새로운 한 태스크가 사용률이 높은 s_1의 실행 큐에 할당하는 것이 타당하게 보인다. 그 이유는 s_1의 시행 큐 부하가 s_2의 실행 큐 부하보다 작기 때문이다.

하지만, s_1에 태스크를 할당하는 것은 (L, H) 상태로 코어의 상태가 바뀌는 확률을 높이게 된다. 이는 governor가 다음 governor epoch 이후의 동작 주파수를 증가시켜 에너지 사용을 증가시키게 된다.

제 3 절 해결책

본 절은 빅리틀 아키텍처의 CPU간 이주 모드에 적합한 사용률인지 기반 부하분산 알고리즘(Utilization-aware Load Balancing)을 제안한다. 이 알고리즘은 추정기(estimator)를 사용한다. 어떤 태스크가 시스템의 준비 큐에서 빠져 나와 실행 가능한 상태가 되면 새로 진입하게 될 실행 큐를 선정하게 된다. 이때, 추정기의 역할은 어떤 실행 큐로 할당을 해야 에너지 사용에 가장 적게 영향을 미치는지 계산한다. 본 연구에서는 이 시점을 부하분산을 수행하는 시점으로 정의한다. 또한, 두 가지 추정기를 사용하여, 알고리즘에 각각 적용하고 그 실효성을 검증하였다.

하기 절의 3.1은 사용률인지 기반 부하분산 알고리즘의 전체적인 설명을 하고 3.2와 3.3은 알고리즘에 사용된 두 가지 추정기에 대한 내용을 기술하였다.

3.1 사용률인지 기반 부하분산 알고리즘

ALGORITHM 1은 본 연구에서 제안하는 사용률인지 기반 부하분산 알고리즘(Utilization-aware load balancing)의 의사코드를 나타낸다. 이 알고리즘은 두 단계의 스텝으로 이루어져있다. 첫째, 라인 5~7에 나타난 것처럼, 가장 부하가 큰 실행 큐로부터 태스크를 빼내어 가장 부하가 작은 실행 큐로 옮긴다. 이는 CFS의 기본 철학인 가중치 기반 부하분산 정책과 동일한 방식이다. 본 연구에서는 Linux CFS의 주기적 부하분산 정책과 유휴상태 부하분산 정책을 비활성화 시켰다. 따라서 이 부분이, 시스템이 가상 CPU간 부하분산을 수행하는 유일한 시점이 된다.

둘째, 시스템의 준비 큐에서 빠져 나온 태스크를 할당할 새로운 실행 큐를 선정한다. 이때, 추정기를 사용하며, 그림 7에 표현된 상태도의 동작을 고려한다. 그리고 주파수 상승이나 빅코어로 스위칭될 확률이 가장

ALGORITHM 1. UTILIZATION-AWARE LOAD BALANCING

Input: $S = \{vCPU_0, vCPU_1, vCPU_2, vCPU_3\}$

1: Imbalance= 1.25

2: τ = Dequeue(q_{wait})

3: CPU_{idlest} = Argument $s \in S$ of the minimum L_s

4: $CPU_{busiest}$ = Argument $s \in S$ of the maximum L_s

5: **While** ($L_{CPU_{busiest}} > L_{CPU_{idlest}}$ * Imbalance) **do**

6: $Task_{popped}$ = Dequeue($q_{CPU_{busiest}}$)

7: Enqueue($q_{CPU_{idlest}}$, $Task_{popped}$)

8: **END While**

9: CPU_{Util} = Argument $s \in S$ of the minimum $E[U(s,\tau)]$

10: Enqueue ($\boldsymbol{q_{CPU_{Util}}}$, $\boldsymbol{\tau}$)

작은 실행 큐를 찾는다. $E[U(s,\tau)]$는 다음 governor epoch 이후에 태스크 τ가 삽입될 때 코어의 사용률에 대한 예측 값을 나타낸다. 따라서 이 알고리즘은 이 값이 가장 작은 코어의 실행 큐를 결과값으로 제시한다.

그림 9.는 이 추정기가 동작할 때 사용되는 시점을 나타낸 그림이다. τ가 가상 CPU s로 할당될 때, $\Delta T_{(s,k)}$는 k번째 governor epoch인 $GE_{(s,k)}$부터 가상 CPU s가 선정되는 시점인 T_{vCPU}까지의 시간이다. T_{vCPU}는 절대적인 시간이며, 모든 가상 CPU에게 동일하게 발생한다. i와 j가 다를 때, $GE_{(i,k)}$ 는 반드시 $GE_{(j,k)}$와 같지는 않다. 그리고, i와 j가 다를 때, $\Delta T_{(i,k)}$가 $\Delta T_{(j,k)}$와 같지는 않다. 즉, 각 가상 CPU 마다 서로 다른 시점의 governor epoch가 존재한다. $U(s)$는 가상 CPU s의 $[GE_{(s,k)}, T_{vCPU}]$ 동안

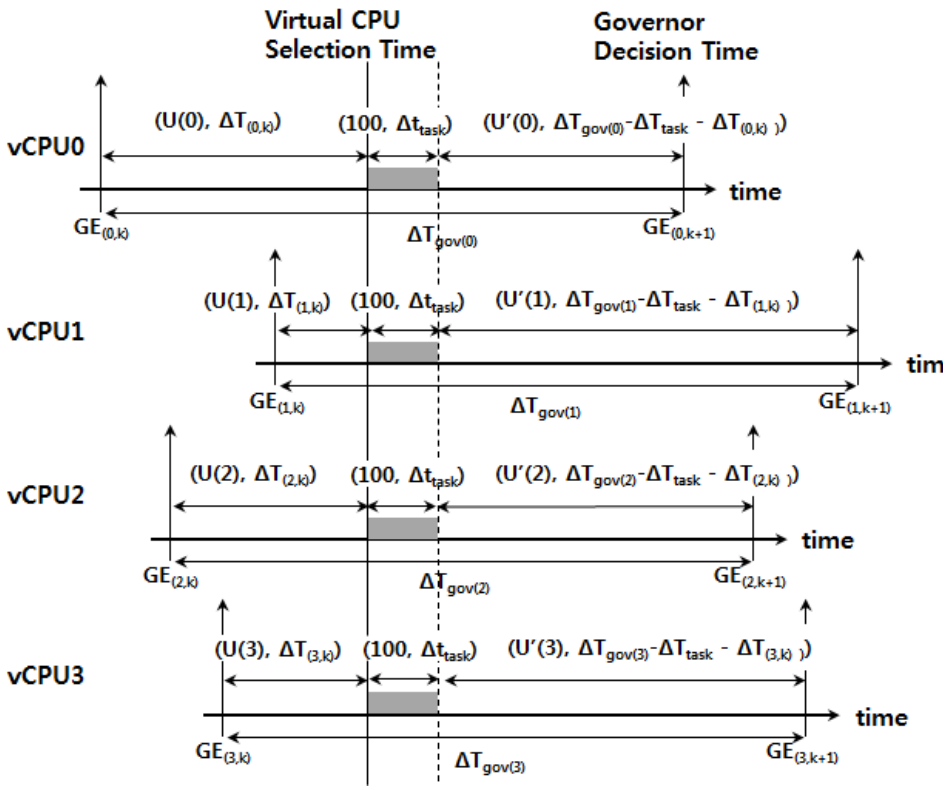


그림 9. 가상 CPU 선택 시점과 Governor Epoch의 관계

측정된 사용률이다. $\Delta T_{gov(s)}$는 가상 CPU s의 governor epoch의 간격을 나타낸다 그리고 $\Delta T_{gov(i)} \approx \Delta T_{gov(j)}$ 을 만족한다. 모든 i 와 j에 대하여, $U'(s)$는 τ가 가상 CPU s로 할당된 후 추정되는 사용률을 나타낸다.

3.2 사용률 기반 추정기

본 연구에서 제안하는 첫 번째 추정기는 다음 사항을 가정한다.

$E_A[U(s,\tau)]$는 첫 번째 추정기가 예측하는 τ가 가상 CPU s로 할당될 때, 다음 governor epoch 이후의 코어 사용률이다. 이 추정기는 τ가 가상 CPU s의 실행 큐에 할당되어도 할당되기 전의 사용률을 그대로 유지할 것이라는 점을 이용한다. 이는 $U'(s) = U(s)$를 뜻한다. 이를 식으로 표현하면 다음과 같다.

$$E_A[U(s,\tau)] = U(s)$$

3.3 수행이력을 반영한 사용률 기반 추정기

두 번째 추정기는 $E_B[U(s,\tau)]$값을 사용하며, 이는τ 가 가상 CPU s로 할당될 때, 예상되는 다음 governor epoch 이후의 코어 사용률이다. 첫 번째 추정기와 달리, 이 추정기는 그림 9에서 $[GE_{(s,k)},\ GE_{(s,k)} + \Delta T_{(s,k)} + \Delta T_{task}]$ 동안의 사용률을 계산한다. 즉, 다시 말하면, T_{vCPU} 시점에 태스크 τ가 가상 CPU s에서 바로 수행되는 것을 가정한다. 그 결과, τ의 수행 시간에 따른 코어 사용률 변동이 일어남을 반영하여 $E_B[U(s,\tau)]$값을 구한다.

이를 위하여, 태스크 τ의 수행시간 정보가 필요하다. 본 연구에서는 이 수행시간 정보를 구함에 있어서, 최근 태스크 τ가 수행했던 16개의 수행시간을 기록하고 이들의 평균을 구한다. 이 값을 ΔT_{sma}라고 표현하고, 이 표현에서 sma는 'Simple Moving Average'의 줄임 말이다. 따라서 $\Delta T_{task} = \Delta T_{sma}$를 뜻하게 된다. τ는 최근 16번 수행될 때, 집합 S에 속한 모든 가상 CPU에서 수행될 수 있기에 ΔT_{sma}는 가상 CPU와 무관하게 단순히 'Simple Moving Average' 방법을 이용하여 구한다. 본 연구에서 제안하는 수행이력을 반영한 사용률 기반 추정기의 결과 값은 다음과 같

다:

$$E_B[U(s,\tau)] = U'(s) = \ U(s) \ \times \frac{\Delta T_{(s,k)}}{\Delta T_{(s,k)} + \Delta T_{sma}} + \frac{100 \times \Delta T_{sma}}{\Delta T_{(s,k)} + \Delta T_{sma}}$$

제 4 장 비대칭 멀티코어 아키텍처용 공정할당 스케줄링

본 장에서는 비대칭 멀티코어 아키텍처에서 사용되는 최적화된 공정할당 스케줄링 기법에 대하여 기술한다. 이를 위하여, 전체 코어 사용 방식인 GTS(Global Task Scheduling) 모드를 지원하는 ARM사의 빅리틀 아키텍처를 대상 시스템으로 정의한다. GTS 모드는 Switcher 모드와 달리 시스템의 모든 빅코어와 리틀코어를 동시에 사용할 수 있다. 따라서 가장 높은 성능을 얻을 수 있어서 개발자들에게 가장 선호되는 빅리틀 시스템의 동작 모드이다.

Linaro는 Linux CFS를 확장하여 GTS모드에 적합한 스케줄링 프레임웍을 제공하고 있다. Linux CFS는 태스크들의 상대적 진척 정도를 virtual runtime로 정의한다. 그리고 태스크들의 virtual runtime을 서로 같게 유지하려고 노력하면서 공정할당 스케줄링을 수행한다. 하지만 공정할당 측면에서 봤을 때, Linaro 스케줄링 프레임웍은 두 가지 문제점이 있다.

첫째, CFS는 단순히 태스크의 가중치에 비례하여 per-core 스케줄링을 수행한다. 어떤 동일한 태스크 τ_i와 τ_j가 각각 빅코어와 리틀코어에서 수행된다고 하면, τ_i의 완료시간이 τ_j보다 당연히 빠를 것이다. 이는 virtual runtime을 통하여 태스크들의 상대적 진척도를 비슷하게 맞추는 CFS의 기본 취지에도 어긋나게 된다. 또한, 같은 코어타입에서 두 동일한 태스크가 수행되더라도, 동작주파수가 큰 코어에서 수행한 태스크가

더 빨리 끝날 것이다. 따라서 비대칭 멀티코어 시스템에서는, 어떤 태스크의 CPU 시간을 계산 시 그 태스크가 사용했던 코어타입과 동작 주파수를 고려해야 한다. 따라서 태스크의 CPU 시간은 코어타입과 동작 주파수에 의하여, 스케일된 CPU 시간으로 계산되어야 한다.

태스크의 스케일된 CPU 시간을 계산 시, 코어의 상태뿐만 아니라 태스크별 수행 특성 역시 고려해야 한다. 이를 예를 들어 설명하면 다음과 같다. 어떤 동일한 태스크 τ_i와 τ_j가 각각 높은 주파수의 빅코어와 낮은 주파수의 리틀코어에서 수행된다고 가정한다. 두 동일한 태스크의 동작 특성은 높은 branch-prediction miss와 큰 메모리 stall을 가지고 있다고 하자.

이러한 경우 τ_i는 빅코어와 높은 주파수의 장점을 다 활용할 수 없다. 하지만, 앞서 설명한 것처럼 스케일된 CPU 시간을 계산시, 코어타입과 동작 주파수만 고려하면, 태스크 τ_i와 τ_j가 진척 정도가 유사함에도 불구하고 태스크 τ_i가 큰 스케일된 CPU 시간을 기록하게 될 것이다.

이 모두를 종합적으로 정리하면 다음과 같은 결과를 나타낸다. 태스크의 스케일된 CPU 시간 산정은 (1)코어타입, (2)동작 주파수, (3)태스크의 동작 특성 등을 모두 고려해야 하고 이들을 per-core 스케줄링에 반영해야 한다.

둘째, CFS는 태스크들의 상대적 진척 정도를 per-core 기반으로 유지하고, 이를 전체 코어로 확대하지 못한다. CFS는 코어 전체에 걸쳐, 모든 태스크에 대하여 virtual runtime을 동일하게 유지하려고 한다. 이를 위해서 코어간 부하분산을 수행한다. 이때 사용하는 부하분산 정책은 태스크의 가중치에 기반을 두고 있다. 즉, 코어당 할당된 태스크들의 가중치 합을 동일하게 유지시키기 위하여 코어간 부하분산을 수행한다. 하지만, 이는 태스크간 상대적 진척도인 virtual runtime을 동일하게 유지하는 것

에 실효성 있는 해결책을 주지 못한다.

본 장에서는 이러한 문제점들을 해결하는 공정할당 스케줄링 기법을 제안한다. 1절에서는 대상 시스템을 정의하고, 현재 사용되는 비대칭 멀티코어 아키텍처용 소프트웨어를 설명한다. 2절에서는 공정할당을 정형화해서 정의하고 3절에서는 해결하고자 하는 문제를 정의한다. 마지막으로 4장에서는 정형화된 문제의 해결책을 제시한다.

제 1 절 시스템 정의

본 절에서는 공정할당 스케줄링 알고리즘이 대상으로 하는 비대칭 멀티코어 아키텍처를 정의하고, 본 학위논문의 이후 부분에서 사용되는 수학적 기호 및 용어를 정리한다. 이어, 대상 시스템으로 사용하는 빅리틀 아키텍처의 GTS 모드를 위하여 개발된 소프트웨어 프레임웍을 설명한다.

1.1 대상 시스템 모델링 및 용어 정리

본 학위논문의 이후 기술된 내용을 이해하는 데 도움을 주기 위하여, 자주 사용되는 수학적 용어들의 의미를 분명하게 하고자 한다. 이들을 표 3에 정리하였다. 이들 수학적 용어들은 제안된 기법의 핵심 기능들을 정형화하여 표현하기 위해서 사용한다. 본 절에서는 공정할당 스케줄링의 대상 시스템인 ARM사의 빅리틀 아키텍처를 표 3에 나타낸 기호를 사용하여 모델링 한다.

그림 10은 본 연구의 대상 시스템을 간략하게 모델링한 그림이다. 대

상 시스템은 한 개의 빅클러스터와 한 개의 리틀클러스터로 이루어져 있고, 각 클러스터는 동일한 r 개의 빅코어들과 s 개의 동일한 리틀코어로 구성된다. 이를 나타내면, 빅클러스터는 $P_b = \{p_1^b, p_2^b, \dots, p_r^b\}$, 리틀클러스터는 $P_l = \{p_1^l, p_2^l, \dots, p_s^l\}$ 로 표현된다. 그리고 모든 코어들은 GTS 모드로 동작한다.

표 3. 수학적 기호 및 용어 설명

Symbol	Definition
r	Number of big cores in the system
s	Number of little cores in the system
$P_b = \{p_1^b, p_2^b, \dots, p_r^b\}$	Set of big cores
$P_l = \{p_1^l, p_2^l, \dots, p_s^l\}$	Set of little cores
$Q_b = \{q_1^b, q_2^b, \dots, q_r^b\}$	Set of run-queues for big cores
$Q_l = \{q_1^l, q_2^l, \dots, q_s^l\}$	Set of run-queues for little cores
n	Number of tasks in the system
$T = \{\tau_1, \tau_2, \dots, \tau_n\}$	Set of tasks in the system
$w(\tau_i)$	Weight of task τ_i
t	Wall clock time
λ	Balancing period
$R_i(t)$	relative performance of a task τ_i at time t
$\hat{c}_i(t)$	Scaled CPU time of task τ_i
$\hat{v}_i(t)$	SVR of τ_i at time t
$\lvert\hat{v}_{i,j}(t)\rvert$	SVR difference between τ_i and τ_j at time t
$\hat{v}_{max}(t)$	The biggest $\lvert\hat{v}_{i,j}(t)\rvert$ at time t
$\Delta\hat{v}_i(r)$	SVR increment of τ_i during r^{th} load balancing period $[(r-1)\lambda,\ r\lambda]$
$\Delta\hat{v}_{i,j}(r)$	Difference between $\Delta\hat{v}_i(r)$ and $\Delta\hat{v}_j(r)$
$w_{\max}/w_{\min}$	Maximum/minimum weight in the system

한 개의 빅코어 $p_i^b \in P_b$는 한 개의 실행 큐 q_i^b를 가진다. 이들 실행 큐들은 한 개의 집합으로 나타내면, $Q_b = \{q_1^b, q_2^b, \ldots, q_r^b\}$로 나타낼 수 있다. 마찬가지로, 리틀코어들은 실행 큐 집합인 $Q_l = \{q_1^l, q_2^l, \ldots, q_s^l\}$를 가진다. 또한, 빅코어와 리틀코어가 운용 중에 취할 수 있는 동작 주파수 집합을 각각 $F(P_b)$와 $F(P_l)$로 나타낸다.

이러한 클러스터 구조로 이루어진, 빅리틀 아키텍처 위에서 운용되는 태스크 모델은, n개의 태스크로 이루어진 하나의 집합으로 나타낸다. 그리고, 이 집합은 $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$로 표현된다. 어떤 태스크 $\tau_i \in T$의 가중치(weight)는 고정된 상수이며 $w(\tau_i)$로 나타낸다. 모든 집합 T에 속한 태스크들은 Linaro 스케줄링 프레임웍이 사용하는 Linux CFS에 의하여 스케줄링 된다. 따라서, 매 스케줄링 tick마다 per-core 스케줄링을 수

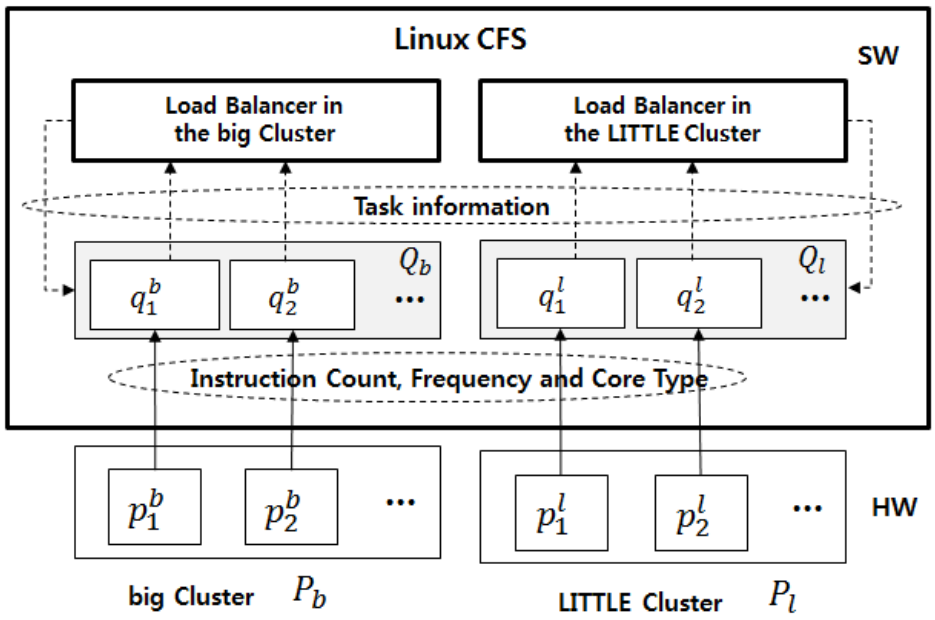


그림 10. 대상 시스템 모델링

행하며, 각 클러스터별로 가중치 기반 부하분산을 수행한다. 이때, 부하분산은 각 클러스터마다 독립적으로 이루어지며, 부하분산 주기는 λ이다.

성능 비대칭성을 스케줄링 알고리즘에 도입하기 위하여, 본 연구에서는 relative performance를 정의한다. 어떤 태스크 τ_i 의 relative performance는 리틀코어의 최소 주파수로 동작 시 측정된 성능과 현재 코어 상태에서 측정된 성능의 비율을 나타낸다. 이때 리틀코어의 최소 주파수는 f_{min}^{l}로 나타내며, $f_{min}^{l} \in F(P_l)$를 만족한다. 각 태스크의 현재 상태 성능은 런타임에 측정된, 스케줄링 tick 사이에 발생한 인스트럭션 수를 나타낸다. 이를 IPT(instruction counts per scheduling tick)로 표기한다. IPT에 대한 측정 방법과 자세한 설명은 본 장의 4절에서 다룬다. $R_i(t)$는 태스크 τ_i의 시간 t에 측정된 relative performance로 정의한다.

1.2 GTS 모드를 위한 Linaro 스케줄링 프레임웍

ARM사의 빅리틀 아키텍처는 GTS 모드를 지원하기 위한 소프트웨어 프레임웍으로써, Linaro 스케줄링 프레임웍을 사용한다. 이는 그림 11에 나타낸 것처럼, Linux의 CFS를 확장해서 빅리틀 아키텍처에 맞게 재구성한 소프트웨어이다.

비대칭 멀티코어 아키텍처가 주는 장점을 최대한 사용하기 위해서 Linaro 스케줄링 프레임웍은 클러스터간 태스크 이주 정책을 기존 Linux CFS에 추가하였다. 이 정책은 현재 CPU-intensive한 태스크들은 빅코어에서 수행되게 하고 백그라운드 태스크와 같은 나머지 태스크들은 리틀코어를 사용하도록 하는 기법이다. 이로써, 시스템의 성능을 향상시킬 수 있다.

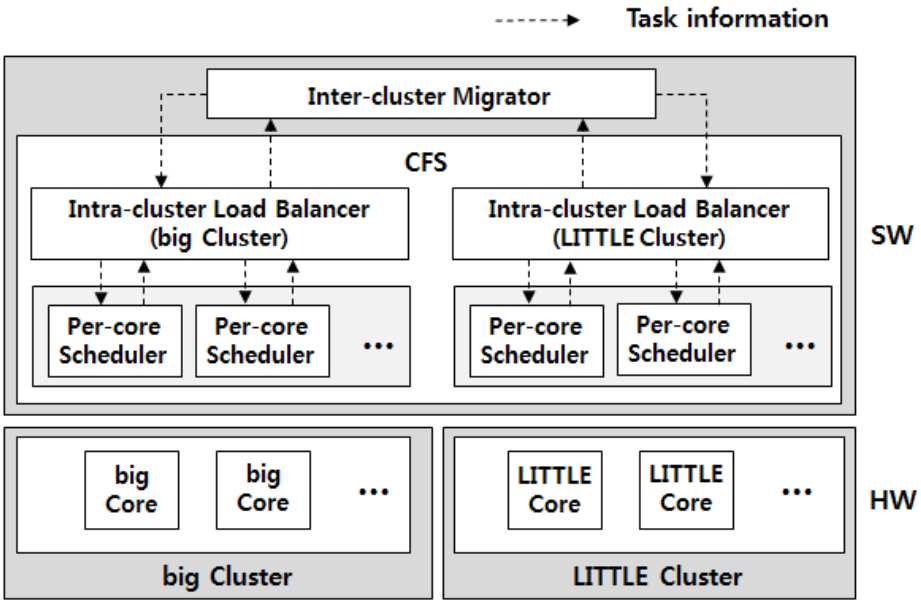


그림 11. Linaro 스케줄링 프레임웍

Linaro 스케줄링 프레임웍은 Linux CFS의 per-core 스케줄링과 가중치 기반 부하분산 정책을 그대로 사용한다. 이때, 부하분산 정책은 가중치를 기반으로 하고, 클러스터 내부에서만 수행된다. 클러스터간 부하분산은 앞서 설명한 클러스터간 태스크 이주정책을 사용한다.

■ **클러스터간 태스크 이주정책**

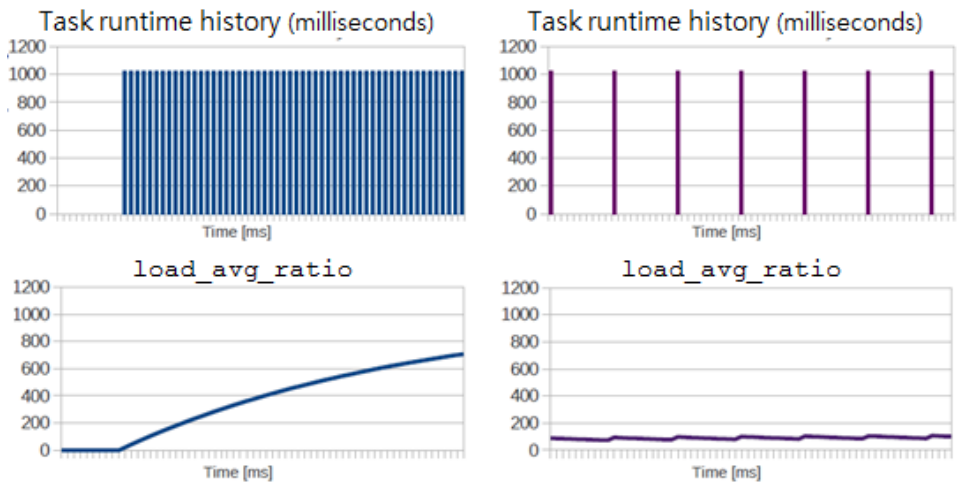
태스크들을 적절한 코어타입에 할당하기 위해서 Linaro 스케줄리 프레임웍은 빅클러스터 P_b 와 리틀클러스터 P_l 사이에 이주정책(Inter-cluster migrator) 을 사용한다. 이를 위하여 각 태스크 별 *load_avg_ratio* 을 주기적으로 체크하며 이는 다음과 같이 정의하고 있다:

$$load\_avg\_ratio = \frac{w_0}{runnable\_avg\_period} \times runanable\_avg\_sum$$

위 식에서 w_0 는 Linux kernel에서 정의하고 있는 nice 값 0 의 가중치를 나타낸다. *runnable_avg_period* 는 해당 태스크의 전체 life time을 나타내고, *runnable_avg_sum*은 태스크가 실행상태로 설정된 시간들을 최근 시간에 가중치를 두어 계산한 값을 나타낸다. 어떤 태스크의 *runnable_avg_sum*을 계산하기 위해서, 실행 가능한 상태로 되어졌던 시간들을 1ms 크기의 segment로 잘게 나눈다. 이 segment들이 최근 시간에 가까울수록, 큰 값을 갖게 된다 [42]. 따라서, 어떤 태스크가 최근에 실행 가능한 상태로 오랜 시간 존재했다면, 이 태스크의 *load_avg_ratio* 는 증가한다.

그림 12는 segment와 *load_avg_ratio*의 관계를 나타내고 있다. 그림의 (a)는 최근까지 태스크가 수행했던 시간이 촘촘하게 기록되고 있다. 따라서 그림에서 알 수 있듯이, *load_avg_ratio* 값은 증가하고 있다. 그림의 (b)는 최근까지 태스크가 수행했던 시간이 기록되었으나, 그 빈도가 드물게 기록되고 있다. 따라서 *load_avg_ratio* 값은 아주 소폭 증가되다가 다시 작은 값으로 돌아오는 과정을 반복하고 있음을 알 수 있다.

어떤 태스크가 리틀코어에서 동작하고 있을 때, 만일 *load_avg_ratio* 값이 어떤 경계값(up-threshold)보다 크게 되면, 이 태스크는 빅클러스터의 한 코어로 이주된다. 반대로 빅코어에서 동작할 때, *load_avg_ratio* 값이 어떤 경계값(down-threshold)보다 작게 되면 이 태스크는 리틀클러스터의 한 코어로 이주되게 된다.



(a) (b)

그림 12. Segment에 따른 ***load_avg_ratio*** 변화

시스템의 성능은 현재 동작중인 태스크의 동작 속도에 많은 영향을 받게 된다. 시스템 운용시간이 증가하면, 이 동작중인 태스크들은 큰 *load_avg_ratio* 값을 가지게 된다. 따라서 빅코어에 남아 있거나 혹은 리틀코어에서 빅코어로 옮겨진다. 반대로, 백그라운드 태스크처럼 최근 실행 가능한 상태로 있었던 시간이 작은 태스크들은 리틀코어에 남거나 혹은 빅코어에서 리틀코어로 옮겨지게 된다. 이러한 클러스터간 이주 정책으로 인하여 시스템의 성능은 향상되게 된다.

■ **클러스터 내부 부하분산 정책**

태스크들에게 할당될 코어 타입이 위에서 언급한 클러스터간 이주정책에 의하여 결정되면, 클러스터 내부에서의 부하분산 정책(Intra-cluster load balancer)에 의하여 태스크들은 코어에 할당된다.

클러스터 내부에서의 부하분산 정책은 3장 1절에서 설명한 부하분산 정책과 동일하다. 하지만, 식 (2)에 나타낸 Γ_s의 역할은 조금 다르다. Γ_s는 빅코어일 때의 값과 리틀코어일 때의 값 두 종류를 가지고 있다. CPU 간 이주 모드에서는, 서로 다른 Γ_s 값을 갖는 코어들 사이에서 부하분산이 이루어진다. 따라서 식 (2)를 계산할 때 Γ_s가 반드시 필요하다.

이와 다르게 GTS 모드용 Linaro 스케줄링 프레임웍은 동일 코어들로 이루어진 클러스터 내부에 한하여 부하분산을 수행한다. 클러스터 내부 코어들은 Γ_s 값이 모두 같기 때문에, 이 값은 부하분산 정책에 있어서 아무런 의미가 없다.

■ Per-core 스케줄링

클러스터내부의 부하분산정책에 의하여 태스크의 코어가 결정되면, Linux CFS는 per-core 스케줄링을 수행한다. CFS는 분산 실행 큐 알고리즘이며, 이는 각 코어에 지정된 실행 큐를 사용한다 [23]. 각 실행 큐는 virtual runtime이 작은 순서대로 정렬되어 있는 실행 가능한 태스크들을 관리한다.

Linux CFS는 이 태스크들의 virtual runtime을 비슷하게 유지하면서 per-core 공정할당 스케줄링을 수행한다. 태스크의 virtual runtime은 다음의 식으로 정의된다.

$$v_i(t) = \frac{w_0}{w(\tau_i)} \times c_i(t) \quad (4)$$

$c_i(t)$ 는 시간 t동안 태스크 τ_i 가 받은 CPU 시간 을 나타낸다. 위 식에서 나타낸 것처럼, 태스크의 virtual runtime은 해당 태스크의 가중치에 반비례하며, CPU 시간에 비례한다.

제 2 절 공정할당의 정의

비대칭 멀티코어 시스템의 공정성(fairness)의 개념을 유도하기 위하여, 우선 현재 사용되는 대칭 멀티코어 시스템에서의 공정성의 정의를 살펴본다 [1][8].

태스크 집합 T를 실행시키고 있는 대칭 멀티코어 시스템을 생각한다. $c_i(t)$ 는 시간 t 동안 태스크 $\tau_i \in T$ 가 받은 CPU 시간을 나타낸다. 대칭 멀티코어 시스템에서 완벽한 공정할당 스케줄러는 다음과 같이 정의한다 [1][8].

정의 1. 구간 $[0, t]$에서 지속적으로 수행 가능한 태스크 $\tau_i \in T$와 $\tau_j \in T$가 존재할 때, 대칭 멀티코어 시스템에서의 완벽한 공정할당 스케줄러는 다음을 만족한다.

$$\frac{c_i(t)}{c_j(t)} = \frac{w(\tau_i)}{w(\tau_j)} \tag{5}$$

이어, 스케일된 CPU 시간을 정의 한다. 태스크 집합 T를 실행시키고 있는 비대칭 멀티코어 시스템을 고려한다. $\hat{c}_i(t)$ 는 시간 t 동안 태스크 τ_i 가 받은 CPU 시간을 나타낸다. 스케일된 CPU 시간은 아래와 같이 정의된다.

정의 2. 비대칭 멀티코어 시스템에서 태스크 τ_i 가 구간 $[0, t]$에서 받은 스케일된 CPU 시간은 다음과 같이 나타낸다.

$$\hat{c}_i(t) = \int_0^t R_i(t)dt \tag{6}$$

이 정의를 기반으로, 우리는 비대칭 멀티코어 시스템에서의 완벽한 공정할당 스케줄러를 다음과 같이 정의한다.

정의 3. 구간 $[0, t]$에서 지속적으로 수행 가능한 태스크 τ_i와 τ_j가 존재할 때, 비대칭 멀티코어 시스템에서의 완벽한 공정할당 스케줄러는 다음을 만족한다.

$$\frac{\hat{c}_i(t)}{\hat{c}_j(t)} = \frac{w(\tau_i)}{w(\tau_j)} \tag{7}$$

본 학위논문에서 제안하는 접근법은 Linux CFS의 per-core 공정할당 스케줄링에 기반을 두고 있고, 이는 virtual runtime을 사용한다. 따라서 우리는 근본적인 CFS의 virtual runtime의 개념을 수정하여 성능 비대칭 특성을 스케줄링에 반영한다. 이어서, 스케일된 virtual runtime을 아래와 같이 정의한다.

정의 4. 시간 t 에서 태스크 τ_i 의 스케일된 virtual runtime SVR(Scaled Virtual Runtime)은 다음과 같다.

$$\hat{v}_i(t) = \frac{1}{w(\tau_i)} \times \hat{c}_i(t) \tag{8}$$

만일 비대칭 멀티코어 시스템에서 사용되는 SVR 기반 스케줄러가 모

든 태스크의 SVR을 같게 한다면 위 식(7)을 만족하기에 완벽한 공정할당 스케줄러라고 할 수 있다.

제 3 절 문제 정의

비대칭 멀티코어 시스템의 모든 태스크들이 임의의 시간 시간 t 에서 동일한 SVR 값을 갖는다면, 이 시스템은 완벽한 공정할당 특성을 나타낸다고 말 할 수 있다. 하지만 실제 상황에서 이러한 스케줄러를 구현한다는 것은 비 현실적이다. 따라서, 본 학위논문에서는 이러한 스케줄러에 기능적으로 근접한 스케줄러를 제안한다. 구체적으로, 태스크간 SVR 값 차이가 상수 값 이내로 제한되게 함으로써 이를 구현한다.

이러한 스케줄러를 구현함에 있어서, 코어들 사이에서 태스크들이 이주되는 것은 필연적으로 발생한다. 그리고 앞서 설명한대로, 빅리틀 아키텍처에서는 두 가지 이주정책이 있다: (1) 클러스터간 태스크 이주정책과 (2) 클러스터 내부 부하분산정책. GTS 모드를 위한 Linaro의 스케줄링 프레임웍은 클러스터간 태스크 이주정책을 사용한다. 이는 태스크의 현재 상태를 파악한 후, 알맞은 코어 타입을 태스크에 할당하여 성능을 최대한 끌어올리기 위함이다.

본 학위논문에서는 성능을 떨어뜨리지 않는 범위에서 태스크간 SVR 차이를 상수 값 이내로 제한함을 목적으로 한다. 이를 위하여, 클러스터간 태스크 이주정책은 원래의 정책을 그대로 사용하고, 클러스터 내부 부하분산 정책을 개선하여, 각 클러스터 내부의 태스크간 SVR 차이를 상수 값 이내로 제한한다.

어떤 클러스터에 존재하는 모든 태스크가 태스크 집합 T 에 속하고, 그들 중 임의의 두 태스크 τ_i 와 τ_j가 선택되었다고 하자. 이때 두 태스

크의 SVR 차이를 $\left|\hat{v}_{i,j}(t)\right|$로 정의한다. 이어서, $\hat{v}_{max}(t)$를 다음과 같이 정의한다:

$$\hat{v}_{max}(t) = \max_{\{\tau_i,\tau_j\}\in T}\left(\left|\hat{v}_{i,j}(t)\right|\right) \leq C \quad (9)$$

위 식을 이용하여 구현하고자 하는 스케줄러의 목적을 설명하면, 각 클러스터의 $\hat{v}_{max}(t)$가 상수 C보다 작은 값으로 제한되게 하는 것이다.

제 4 절 해결책

본 장에서는 앞 절에서 정의된 문제를 해결하는 해결책을 제시한다. 본 학위논문에서 제안하는 기법은 크게 두 가지로 구성되어 있다: (1) relative performance를 사용하여 각 태스크의 SVR을 산정한다. (2) 클러스터 내부에 존재하는 실행 가능한 태스크들의 SVR 값을 기반으로, 코어간 태스크들을 이주 시킨다. 이를 통하여 클러스터 내부의 태스크간 SVR 값 차이를 상수 값 이내로 제한한다.

제안된 기법이 기존의 Linaro 스케줄링 프레임웍과 부작용 없이 동작하기 위하여, SVR calculator 라는 새로운 모듈을 각 코어 별로 존재하는 per-core 스케줄러에 추가하였다. 또한, 각 클러스터에 존재하는 가중치 기반 부하분산 정책을 SVR 기반 부하분산 정책으로 교체하였다.

동적으로 변화하는 태스크의 phase 특성을 반영하기 위하여, SVR calculator는 매 스케줄링 tick마다 현재 수행되고 있는 태스크의 relative performance를 측정한다. 본 연구에서 스케일된 CPU 시간과 누적된 형태의 SVR 값을 구하기 위하여, 몇 개의 kernel 함수를 수정하

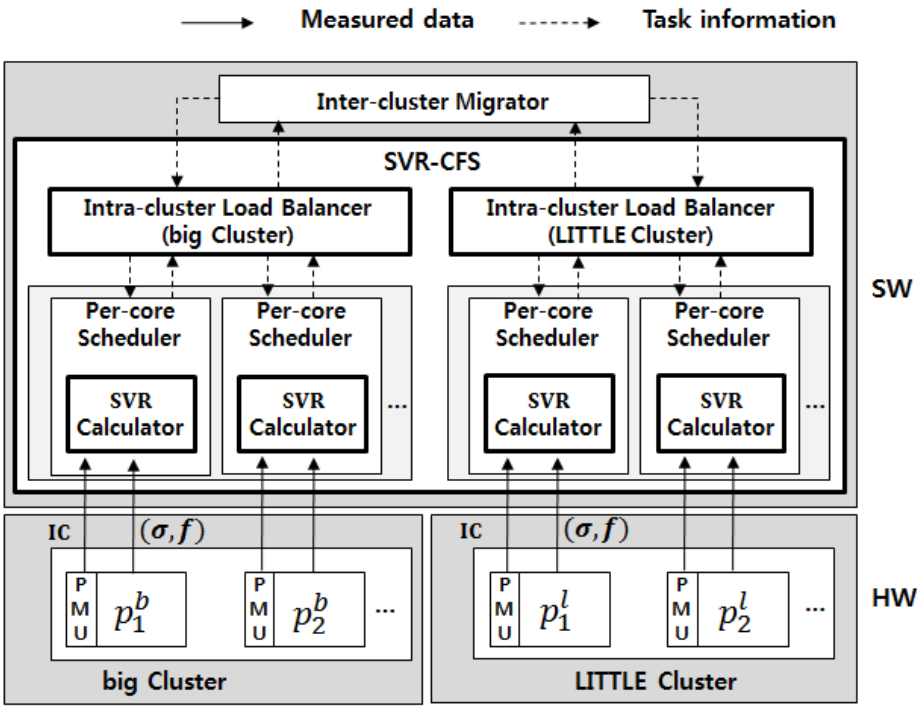


그림 13. 공정할당 스케줄링을 위한 해결책 개괄 도표

였다. SVR 기반 부하분산 기법은 각 클러스터 내부에 존재하며, 같은 클러스터 내부에서 실행 가능한 태스크들의 SVR 값 차이를 상수 값 이내로 제한한다. 본 연구에서는 이렇게 도출된 스케줄러를 SVR-CFS라고 정의한다.

그림 13은 제안하는 기법의 전체적인 아키텍처를 보여준다. 그림에서 알 수 있듯이, 제안된 기법은 클러스터 구조의 빅리틀 멀티코어 시스템 위에서 구현되었다. SVR calculator는 하드웨어로부터 수행된 IC(instruction counts) 값, 코어 타입(σ), 동작 주파수(f)등의 정보를 받는다. 이어서, 현재 수행중인 태스크의 relative performance를 계산한다. 이렇게 계산된 결과를 사용하여, 그 태스크의 SVR 값으로 확장한

다.

이 값을 기반으로, SVR-CFS 각 코어의 실행 큐에 속해있는 실행 가능한 태스크들에 대하여 per-core 스케줄링을 수행한다. 미리 설정된 주기 값을 기반으로 매 주기마다, SVR 기반 부하분산 정책은 클러스터 내부의 태스크들에 대하여 SVR 값 차이가 상수 값 이내로 제한되게 부하를 분산시킨다. 결과적으로 본 논문에서 제안하는 기법은 공정할당 스케줄링을 수행한다.

이어지는 하기 절의 4.1 은 SVR을 구하는 방법을 기술한다. 4.2는 SVR 기반 per-core 스케줄링을 설명하고, 4.3은 SVR 기반 부하분산 정책에 사용된 알고리즘들을 자세히 설명한다. 마지막으로 4.4는 제안된 알고리즘을 수학적으로 분석하고 이를 검증한다.

4.1 SVR 계산

태스크의 SVR을 구하기 위해서는 먼저 그 태스크가 나타내는 relative performance를 구해야 한다. 태스크 τ_i의 $R_i(t)$는 시간 t에서 현재 코어가 나타내는 성능과, 주파수가 $f_{min}^{l} \in F(P_l)$ 상태에서 측정된 성능의 비율을 나타낸다. 앞서 설명한 IPT를 이용하여 이를 정의하면 다음과 같다.

$$R_i(t) = \frac{IPT_{real}(\tau_i, t)}{IPT_{base}(\tau_i, t)} \tag{10}$$

위 식에서, $IPT_{real}(\tau_i, t)$는 시간 t 에서 측정한 실제 IPT를 뜻하고, $IPT_{base}(\tau_i, t)$는 시간 t에서 코어가 $f_{min}^{l} \in F(P_l)$로 동작했음을 가정했을

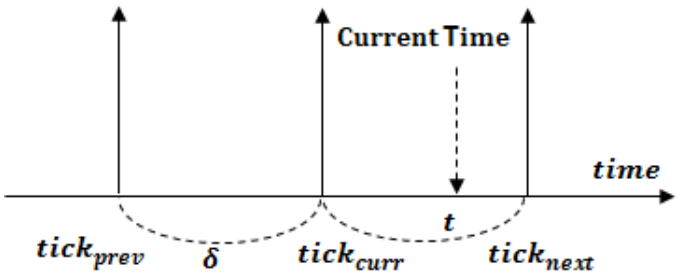


그림 14. 현재시간 t 와 스케줄링 tick의 관계

때의 IPT이다. 이 두 값들 모두 가장 최근 두 개의 스케줄링 tick 사이에서 측정된 IPT를 나타낸다. 현재 시간 t와 스케줄링 tick의 관계를 보다 쉽게 이해하기 위해서, 이를 그림 14에 나타내었다. 가장 최근 두 개의 스케줄링 tick이란 그림에서 볼 수 있는 것처럼 $tick_{prev}$와 $tick_{curr}$를 뜻한다.

그림에서 알 수 있듯이, $R_i(t)$는 매 스케줄링 tick마다 갱신된다. 태스크 τ_i가 최근 두 개의 스케줄링 tick사이에 동작하지 않았다면, $R_i(t) = 0$을 나타낸다. 그 이유는 어떠한 instruction도 코어에서 수행되지 않았기 때문이다.

$IPT_{real}(\tau_i, t)$ 는 그림 13에서 알 수 있듯이, 각 코어 별 PMU(performance monitoring unit)를 통하여 구할 수 있다. PMU는 코어가 수행한 instruction, cache miss등의 정보를 레지스터에 카운터 값으로 저장한 후 알려주는 하드웨어 장치이다 [5][6].

하지만, $IPT_{base}(\tau_i, t)$는 $IPT_{real}(\tau_i, t)$의 경우처럼 바로 구할 수 없다. 그 이유는 태스크 τ_i가 시간 t에서 $f_{min}^{l} \in F(P_l)$로 동작하지 않았을 확률이 높기 때문이다. 이 값을 구하기 위해서 본 연구에서는 간단한 linear

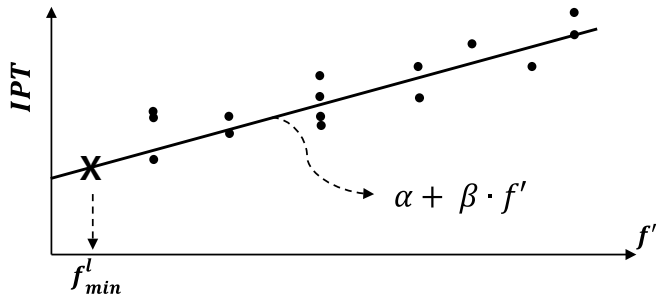


그림 15. IPT 와 동작 주파수 f'

regression 방법을 사용하였다. 사용된 방법은 OLS(Ordinary Least-Squares) regression이다 [43].

$F(P_b)$에 속한 주파수와 $F(P_l)$에 속한 주파수 중 몇 개의 값은 서로 같은 값을 갖는다. 하지만 코어의 성능 차이로 인하여 어떤 태스크의 relative performance는 주파수가 같더라도 다르다. 따라서, 빅코어와 리틀코어의 주파수 값들이 서로 겹치지 않고 하나의 도메인에서 표현되도록 아래와 같은 식으로 나타낸다.

$$f' = \varepsilon \cdot f$$

위 식에서 f는 코어의 동작주파수를 나타내고, ε는 미리 정해진 상수를 나타낸다. 본 연구에서 ε는 동작주파수가 $f \in F(P_l)$일 경우 1의 값을 갖고 $f \in F(P_b)$일 경우 1.8을 갖게 하였다. 이 값들은 그림 16을 기준으로 산정한 값이며 [44], 그림에 대한 자세한 설명은 이 세부 절의 마지막 부분에 기술하였다.

이렇게 얻어진 f' 와 태스크의 IPT 데이터를 이용하여 OLS

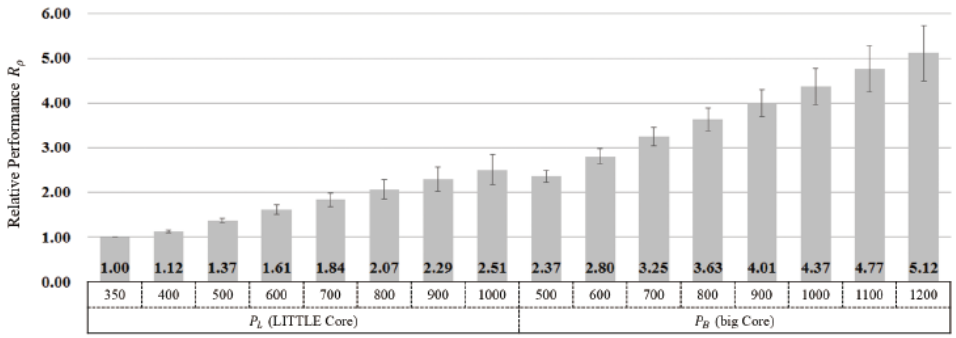


그림 16. 동작 주파수 f'에 따른 R_ρ

regression를 수행하는 방법은 다음과 같다. 첫째, f'에 따른 어떤 태스크의 IPT 데이터를 최근 k 스케줄링 tick동안 저장한다. 이때, k값은 본 연구에서 20으로 설정하였다. 이어, 다음의 식이 나타내는 OLS regression의 α, β 값들을 구한다 [43].

$$IPT = \alpha + \beta \cdot f' \tag{11}$$

마지막으로 위 식의 f'에 f_{min}^{l}값을 대입함으로써 $IPT_{base}(\tau_i, t)$값을 구한다. 이를 그림 15에 나타내었다.

시스템이 $R_i(t)$값을 구하기 위한 충분한 양의 IPT와 f' 데이터가 없을 경우가 발생할 수 있다. 즉, 해당 태스크가 시작한지 얼마 되지 않았거나, 혹은 한 개의 고정된 주파수로 동작했을 때이다. 이러한 경우, SVR calculator는 시스템이 수행되기 전에 미리 얻어진 R_ρ 값을 사용한다.

그림 16는 R_ρ와 f'의 관계를 나타낸다. 이를 위하여 본 연구에서는 SPEC CPU2006 벤치마크 [45][46] 중 bzip2, xalanc, h.264ref, sjeng을 사용하고, 이들의 각 f'에 따른 완료시간 평균을 구하였다. $f' = f_{min}^{l}$에

서 4가지 벤치마크의 평균 완료시간을 $R_\rho = 1$ 로 정의하고, 나머지 주파수 f' 에서 측정된 평균 완료시간들은, 이 값을 기준으로 normalize한 값이다 [44].

4.2 SVR 기반 per-core 스케줄링

앞서 설명한 태스크들의 SVR 값들에 기반하여, SVR-CFS는 per-core 공정할당 스케줄링을 수행한다. 태스크들은 그들에게 부여된 time slice를 다 소진했을 경우, 현재 수행된 코어의 실해 큐로 돌아가게 된다. 이때, 실행 큐 내부에서의 위치는 태스크의 SVR 값에 의하여 결정된다.

SVR-CFS역시 원래의 CFS와 마찬가지로 SVR 크기 순서로 정렬된 red-black 트리를 사용한다. 따라서 실해 큐로 복귀하는데 걸리는 시간은 $O(log\, n)$ 이며, n은 트리에 속해있는 태스크의 수이다. SVR-CFS가 다음 수행시킬 태스크를 찾을 때는 실행 큐의 가장 작은 SVR 값을 갖는 트리의 노드를 찾으면 된다. SVR-CFS는 트리의 가장 아래쪽에 위치한 노드들 중 가장 왼쪽에, SVR값이 가장 작은 태스크 노드를 할당한다. 따라서 실행 큐로 복귀할 때와는 대조적으로 이때의 수행시간은 $O(1)$이다.

본래의 CFS에서는, 태스크의 virtual runtime(virtual runtime)을 관리할 때, 누적된 형태로 관리하지 않는다. CFS는 부하분산을 위하여 어떤 태스크가 실행 큐로 진입하거나 빠져나갈 때, 원래의 virtual runtime을 갱신한다. 그 이유는, 실행 큐내부에서 태스크들이 virtual runtime을 취할 때 상대적인 값을 갖게 하기 위해서이다 [23][24].

시간 t_1에서 태스크 τ_i가 실행 큐 q_j에서 빠져나올 때, 이 태스크의

virtual runtime 은 본래의 CFS에서 아래 식처럼 계산된다.

$$v_i^{'}(t_1) = v_i(t_1) - v_{min}^{j}(t_1) \quad (12)$$

위 식에서, $v_{min}^{j}(t_1)$은 시간 t_1에서 태스크 τ_i가 실행 큐 q_j에 존재할 때, q_j에 속해있는 태스크들 중 가장 작은 virtual runtime을 갖는 태스크의 virtual runtime이다.

반대로, 시간 t_2에서 태스크 τ_i가 실행 큐 q_j에 삽입될 때, 이 태스크의 virtual runtime은 아래 식처럼 본래의 값 대신 갱신된다.

$$v_i(t_2) = v_i^{'}(t_2) + v_{min}^{j}(t_2) \quad (13)$$

위 식들을 쉽게 이해하기 위해서, 그림 17에 CFS의 virtual runtime 관리 기법을 나타내었다. 왼쪽의 그림은 시간 t_1에서 'Task 1'이 실행 큐 q_0에서 빠져나올 때, $v_{min}^{0}(t_1)$ 값이 1000에서 0으로 바뀌는 과정을 설명하고 있다. 이렇게 바뀐 이유는, 위 식 (12)에 나타내었듯이, q_0의 최소 virtual runtime이 'Task 1' 자신 본래의 virtual runtime이기 때문이다.

오른쪽 그림은 위와 반대로, virtual runtime 값이 50인 'Task 3'이 q_1에 삽입될 때의 동작을 나타낸다. 위 식 (13)에 나타내었듯이, q_1의 최소 virtual runtime이 900이기 때문에, 50에서 950으로 갱신되어 삽입된다.

이처럼 본래의 CFS는 virtual runtime을 계산할 때, 어떤 태스크의 모든 과거 수행시간 이력을 보존하지 않는다. 따라서, 이러한 방법은 본 학위논문에서 제안하는 기법에 적용할 수 없다.

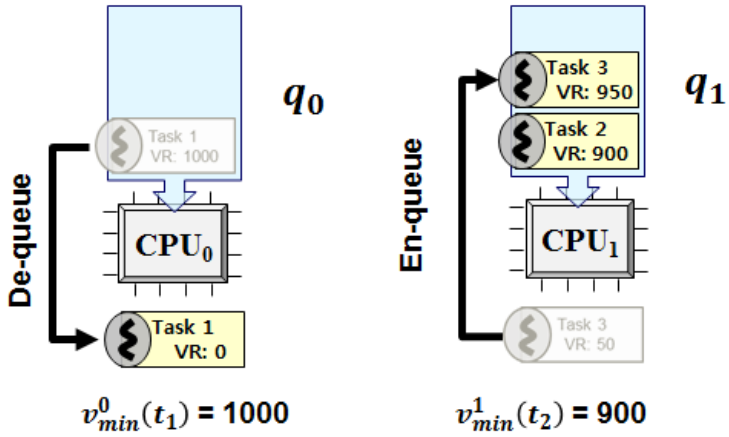


그림 17. CFS의 virtual runtime 관리 기법

정의 2 에 기술된 것 처럼, 태스크 τ_i의 스케일된 CPU 시간을 구하기 위해서는 이 태스크의 모든 수행시간 이력을 보존해야한다. 식 (6)에 나타낸 것처럼, $\hat{c}_i(t)$는 태스크 τ_i의 $[0,t]$ 동안의 수행 이력을 가지고 있다. 이는 태스크가 생성된 후부터 시간 t까지 모든 부분 구간 수행 이력을 가지고 있다. 또한, 이는 각 부분 구간 동안, 그 태스크에 할당된 동작 주파수, 코어타입, 동작특성 등을 모두 가지고 있다. 따라서, 본 연구에서는 CFS의 virtual runtime 갱신 방식을 수정하여, SVR-CFS는 시스템의 모든 태스크들의 SVR을 누적된 형태로 관리하도록 하였다.

4.3 SVR 기반 부하분산 알고리즘

본 절에서는 같은 클러스터 내부의 어떤 태스크 쌍도 그 SVR 차이가 상수 값 이내로 제한되는 SVR 기반 부하분산 알고리즘에 대하여 기술한

다. 제안된 알고리즘은 빅클러스터 및 리틀클러스터의 구분 없이 적용되는 알고리즘이다. 따라서, 알고리즘 설명에 필요한, 표 3에 정의된 각 클러스터별 수학적 기호를 단순화 시킬 필요가 있다. 이를 클러스터 구분 없이 단순화하여 표현하면 다음과 같다.

m개의 동일한 코어로 이루어진 임의의 클러스터는 n개의 실행 가능한 태스크로 이루어진 태스크 집합 $T = \{\tau_1, \tau_2, \dots, \tau_n\}$를 수행하고 있다. 이때, 이 클러스터의 실행 큐 집합과 코어별 실행 큐는 $Q = \{q_1, q_2, \dots, q_m\}$로 나타낸다. 또한, 각 실행 큐 $q_k \in Q$는 태스크 그룹 G_k를 가지고 있다.

자세한 알고리즘의 설명에 앞서, feasible set을 정의한다. 이는 멀티코어 아키텍처의 공정할당 스케줄러가 스케줄링이 가능한 태스크 집합을 뜻한다. 본 연구에서 feasible set의 정의는 다음과 같다. 한 태스크 집합 $T = \{\tau_1, \tau_2, \dots, \tau_n\}$가 m개의 코어로 이루어진 클러스터 내부에서 운용될 때, 다음 조건을 만족하면 feasible set이라고 부를 수 있다 [23].

$$\forall i: 1 \le i \le n, \qquad w(\tau_i) \le \frac{\sum_{j=1}^{n} W(\tau_j)}{m} \tag{14}$$

위 식의 의미는 다음과 같이 설명할 수 있다. 어떤 클러스터 내부의 어떤 태스크 τ_i의 가중치는 클러스터내부에 존재하는 모든 태스크들의 가중치 합의 $1/m$ 크기보다 작거나 같아야 한다는 뜻이다. 본 연구에서 제안하는 기법은 이를 만족하는 태스크 집합 $T = \{\tau_1, \tau_2, \dots, \tau_n\}$에 대해서 동작한다.

본 논문의 이후에 기술되는 내용의 이해를 돕고자 어떤 태스크의 스케일된 가중치와 어떤 태스크 그룹의 스케일된 부하를 정의한다.

정의 5. 어떤 태스크 τ_i의 r번째 부하분산 주기$[(r-1)\lambda,\ r\lambda]$ 동안의

스케일된 가중치는 다음과 같다.

$$\widehat{w}(\tau_i, r) = \frac{w(\tau_i)}{R_i^{avg}(r)}$$

위 식에서 $R_i^{avg}(r)$는 τ_i 가 $[(r-1)\lambda,\ r\lambda]$ 동안 동작할 때, 평균 relative performance $R_i(t)$ 값을 나타낸다.

정의 6. 어떤 태스크 그룹 G_k의 r번째 부하분산 주기$[(r-1)\lambda,\ r\lambda]$ 동안의 스케일된 부하는 다음과 같다.

$$\hat{L}_k(r) = \sum_{\tau_i \epsilon G_k} \widehat{w}(\tau_i, r)$$

대칭 멀티코어 아키텍처에서는 각 태스크에게 주어지는 CPU시간은 단순하게 가중치에 비례한다. 이와 달리, 비대칭 멀티코어 아키텍처에서는 CPU 시간은 코어의 컴퓨팅 능력이 반영된 가중치에 비례해야 한다. 이를 위하여 본 연구에서 태스크의 스케일된 가중치와 태스크 그룹의 스케일된 부하를 **정의 5** 와 **정의 6**에 나타내었다.

식 (5),(6),(7)을 사용하여 $c_i(t)$를 $w(\tau_i)$와 $R_i(t)$로 나타내면, 보다 직관적으로 알 수 있다. 이로써 비대칭 멀티코어 아키텍처의 완벽한 공정 할당 스케줄링을 위해서는 어떤 태스크 τ_i의 CPU 시간 $c_i(t)$는 매 부하분산 주기 동안 스케일된 가중치에 비례해야 함을 알 수 있다. 본 학위논문의 이후에서는, 수학적 용어 표현을 간단히 하기 위해서 $\hat{L}_k(r)$를 $\hat{L}_k$로 간략히 표기한다.

ALGORITHM 2. SVR-BASED LOAD BALANCING

Input: A set of tasks in a cluster: T
The number of cores: m

1: H = **SORT**(T)
2: $G \leftarrow$ **SPLIT**(H, m)
3: $G \leftarrow$ **ADJUST**(G)
4: **return** G

ALGORITHM 2는 top-level SVR-BASED LOAD BALANCING 알고리즘을 나타내고 있다. 이 알고리즘은 매 부하분산 주기 λ마다 불려지고, 그때마다 각 태스크들의 실행 큐를 결정한다. 알고리즘의 입력으로 두 가지 인자를 받게 된다: (1) 어떤 클러스터 내부의 실행 가능한 태스크 집합 T와 (2) 클러스터가 가지고 있는 코어 수 m. 이 알고리즘은 m개의 태스크 그룹 $G = \{G_1, G_2, \ldots, G_m\}$를 출력하고, 다음의 성질을 만족한다.

(1) $1 \le k < m$를 만족하는 모든 k에 대하여, 태스크 그룹 G_k에 속한 모든 태스크의 SVR 값은 G_{k+1}에 속한 어떠한 태스크의 SVR 값보다 작거나 같다.

(2) $\hat{L}_1 \le \hat{L}_2 \le \cdots \le \hat{L}_m$.

(3) $0 \le \hat{L}_{k+1} - \hat{L}_k \le 2w_{max}$.

최종 출력된 모든 G_k에 대하여, 속한 태스크들은 실행 큐 q_k에 할당된다. 그리고 이러한 $G = \{G_1, G_2, \ldots, G_m\}$이 존재하는 경우, 태스크 집합 T를 balanceable set이라 부른다.

이 알고리즘은 먼저 라인 1에 나타내 것처럼 merge-sort를 수행한다. 이때, SVR 값들의 오름차순으로 각 태스크들을 정렬한 후 시퀀스 H를 생성한다. 그 후, SPLIT을 수행한다 (라인 2). 이는 정렬된 시퀀스 H를 m 태스크 그룹으로 나눈다. 이때 생성된 태스크 그룹의 집합은 $G = \{G_1, G_2, \ldots, G_m\}$이며, 각 인접한 태스크 그룹간의 스케일된 부하 차이는 상수 값 이내로 제한된다. 라인 3에 기술된 ADJUST는 라인 2에 기술된 SPLIT 의 결과인 $G = \{G_1, G_2, \ldots, G_m\}$를 입력으로 받는다. 그리고 이를 조정하여, 위에 기술된 SVR-BASED LOAD BALANCING의 세 가지 성질을 만족하는 $G = \{G_1, G_2, \ldots, G_m\}$를 최종 출력하게 된다.

앞서 기술된 SVR-BASED LOAD BALANCING의 성질 중 (1)과 (2)는 다음을 만족한다. 큰 SVR 값을 가진 태스크들은 더 큰 스케일된 부하 값을 가지는 코어에서 수행되므로, 다음 부하분산 주기 시점까지 더 느리게 SVR 값이 증가하게 된다. 또한, (3)은 큰 SVR 값을 가진 태스크들이 작은 SVR 값을 가진 태스크들 보다 너무 느리게 SVR 값이 증가하지 않게 제한을 둔다.

ALGORITHM 3은 SPLIT의 의사 코드를 나타내고 있다. 이는 정렬된 시퀀스 H를 입력으로 받은 후 m개의 태스크 그룹을 생성한다. 이렇게 나뉘어진 m개의 태스크 그룹은 $G_1, G_2, \ldots, G_m$형태로 정렬이 되고, 각 인접한 태스크 그룹의 스케일된 부하의 차이는 상수 값 이내로 제한된다.

ALGORITHM 3. SPLIT

Input: A sequence of tasks sorted in ascending order with SVR: $H =< \tau_1, \tau_2, \ldots, \tau_n >$, The number of cores: m

SPLIT(H, m)

1: $G_1 \leftarrow \emptyset,\ G_2 \leftarrow \emptyset, \ldots,\ G_m \leftarrow \emptyset$
2: $i \leftarrow 1$
3: $\hat{L}_{acc} \leftarrow 0$
4: **for** $k \leftarrow 1$ **to** $m-1$ **do**
5: $\hat{L}_k^E \leftarrow$ **CALCEXPECTEDLOAD**(H, m, $\hat{L}_{acc}$, k)
6: **while** $\hat{L}_k + W(\tau_i) \leq \hat{L}_k^E$ **do**
7: $G_k \leftarrow G_k \cup \{\tau_i\}$
8: $i \leftarrow i+1$
9: **end while**
10: $\hat{L}_{acc} \leftarrow \hat{L}_{acc} + \hat{L}_k$
11: **end for**
12: **return** $\{G_1, G_2, \ldots, G_m\}$

CALCEXPECTEDLOAD(H, m, $\hat{L}_{acc}$, k)

13: $\hat{L} \leftarrow$ sum of scaled weights of tasks in H
14: $\hat{L} \leftarrow (\hat{L} - \hat{L}_{acc})/(m-k+1)$
15: **return** $\hat{L}$

이 알고리즘은 각 태스크 그룹 $G_1, G_2, \ldots, G_m$에 반복적으로 태스크를 할당한다. 이 동작을 수행하기 위해서, 알고리즘은 먼저 라인 5처럼 CALCEXPECTEDLOAD 함수를 호출한다. 이 함수는 태스크 그룹 G_k가 취할 수 있는 스케일된 부하의 기대값을 구하며, 이 기대값은 다음과 같이 정의된다.

$$\hat{L}_k^E = \frac{\sum_{j=k}^{m} \hat{L}_j}{m-k+1} \tag{15}$$

위 식에서 알 수 있듯이, $\hat{L}_k^E$ 는 $m-k+1$ 개의 태스크 그룹인 $G_k, G_{k+1}, \ldots, G_m$에서 각 태스크 그룹에 할당 가능한 스케일된 부하의 평균 값을 나타낸다.

이렇게 구해진 값을 사용하여 $\hat{L}_k$ 가 $\hat{L}_k^E$ 보다 작거나 같을 경우, $G_1, G_2, \ldots, G_{k-1}$에 속하지 않는 태스크 중 가장 작은 SVR 값을 갖는 태스크를 반복해서 G_k에 할당한다 (lines 6~9).

SPLIT 은 $G = \{G_1, G_2, \ldots, G_m\}$에 속한 모든 태스크 그룹에게 비슷한 스케일된 로드가 부여되도록 노력한다. 이러한 이유로 각 인접한 태스크 그룹들간의 스케일된 부하 차이는 상수 값 이내로 제한될 수 있다. 이에 관한 자세한 설명과 증명은 다음 세부 절에서 하기로 한다.

ALGORITHM 4 는 ADJUST 의 의사 코드를 나타낸다. 이 알고리즘의 목적은 $G_1, G_2, \ldots, G_m$ 태스크 그룹들이 갖는 스케일된 부하 값들이 $\hat{L}_1 \le \hat{L}_2 \le \cdots \le \hat{L}_m$ 의 관계를 만족하도록 하는 것이다. SPLIT 의 출력이 ADJUST 의 입력으로 부가되며, 이 또한 m 개의 태스크 그룹 $G = \{G_1, G_2, \ldots, G_m\}$를 출력한다.

이 알고리즘은 반복적으로 CHECKANDMOVETASKS 함수를 호출하며, 이때 입력을 G_{m-1} 에서 G_1 순으로 변화 시킨다 (lines 2~4). 이 함수의 주된 동작은 k 값을 입력으로 받아서 태스크 그룹 $G_k, G_{k+1}, \ldots, G_m$ 의 스케일된 부하의 순서를 $\hat{L}_k \le \hat{L}_{k+1} \le \cdots \le \hat{L}_m$ 로 하게 한다. 따라서 ADJUST 의 모든 동작이 완료된 시점에는 $\hat{L}_1 \le \hat{L}_2 \le \cdots \le \hat{L}_m$ 의 성질을 얻을 수 있다.

CHECKANDMOVETASKS 이 불릴 때 마다, $\hat{L}_k$ 와 $\hat{L}_{k+1}$ 의 대소 관계를 살핀다. 이때 $\hat{L}_k \leq \hat{L}_{k+1}$ 이 만족한다면, 이 함수는 아무런 동작을 수행하지 않고 빠져 나온다. 하지만 $\hat{L}_k > \hat{L}_{k+1}$ 경우에는, G_k 에 속한 태스크 중 가장 큰 SVR 값을 가진 태스크가 G_{k+1} 태스크 그룹으로 $\hat{L}_k \leq \hat{L}_{k+1}$를 만족할 때까지 옮겨지게 된다 (lines 6~13).

이어서, 이 함수는 G_k 와 G_{k+1} 의 사이에서 발생한 태스크 이동이 G_{k-1} 와 G_{k+2} 에 끼친 영향을 조사한다. 이를 위하여 CHECKANDMOVETASKS 함수의 입력으로 $k-1$ 와 $k+1$ 을 부가하여 CHECKANDMOVETASKS 함수를 재귀적으로 호출한다 (lines 14~19). 이렇게 함으로써, k 값을 입력으로 받은 CHECKANDMOVETASKS 함수는 $\hat{L}_{k-1} \leq \hat{L}_k \leq \hat{L}_{k+1} \leq \hat{L}_{k+2}$ 의 성질을 만족 시킨다.

위에서 설명한 ADJUST 가 모든 동작을 완료한 시점에는, 앞서 기술한 SVR-BASED LOAD BALANCING 의 세가지 성질을 만족하게 된다. 이에 대한 자세한 수학적 분석 및 검증은 다음 세부 절에서 설명한다.

ALGORITHM 4. ADJUST

Input: Set of task groups: $G = \{G_1, G_2, \ldots, G_m\}$

ADJUST(G)

1: **for** $k \leftarrow m-1$ **down to** 1 **do**
2: **CHECKANDMOVETASKS**(G, k)
3: **end for**
4: **return** G

CHECKANDMOVETASKS (G, k)

5: **if** $\hat{L}_k > \hat{L}_{k+1}$ **then**
6: **while** $\hat{L}_k > \hat{L}_{k+1}$ **do**
7: **if** $k = 1$ **and** there is only one task in G_1 **then**
8: **print** *"Given task set is not a balanceable set"*
9: **return** $\emptyset$
10: **else**
11: move the task with the largest SVR in G_k to G_{k+1}
12: **end if**
13: **end while**
14: **if** $k-1 \geq 1$ **then**
15: **CheckAndMoveTasks**$(G, k-1)$
16: **end if**
17: **if** $k+1 \leq m-1$ **then**
18: **CheckAndMoveTasks**$(G, k+1)$
19: **end if**
20: **end if**

4.4 알고리즘의 수학적 분석 및 검증

본 절에서는 본 학위논문에서 제안된 SVR-BASED LOAD BALANCING 알고리즘을 자세히 수학적으로 분석한다. 분석함에 있어서 구체적으로, 같은 클러스터 내부에서 임의의 두 태스크는 임의의 시간 t 에서 $\hat{v}_{max}(t)$ 값이 상수 값 이내로 제한됨을 보인다.

이를 위하여 본 절에서는 첫째, 앞서 설명한 SVR-BASED LOAD BALANCING 알고리즘의 세 가지 성질을 증명한다. SPLIT과 ADJUST 를 수행하면서, 클러스터 내부의 모든 태스크들의 SVR 값에 의한 순서는 변하지 않는다. 따라서 (1)은 당연히 만족된다. ADJUST가 가지고 있는 본연의 특징에 의하여 (2) 역시 만족된다. 따라서 본 절에서는 성질 (3)이 만족됨을 보임으로써 증명을 완료한다.

LEMMA 4.1. 동일 클러스터 내부에 존재하는 임의의 인접하고 있는 태스크 그룹의 스케일된 부하의 차이는 $2w_{max}$보다 작거나 같고, 이는 다음의 식으로 표현된다.

$$0 \leq \hat{L}_{k+1} - \hat{L}_k \leq 2w_{max} \text{ for } 1 \leq k < m$$

PROOF. 위 식에 대한 증명은 두 단계의 스텝으로 이루어진다: (a) SPLIT에 의하여 인접한 태스크 그룹간의 스케일된 부하의 차이는 상수 값 이내로 제한된다. 그리고 (b) LEMMA 4.1은 ADJUST에 의하여 만족한다.

Step (a): SPLIT 은 반복적으로 태스크 그룹 G_k에 가장 작은 SVR 값을 갖는 태스크를 삽입 시킨다. 이를 수행함에 있어서, $\hat{L}_k$가 $\hat{L}_k^E$를 넘지 않을

때까지 수행한다. 따라서 다음의 식을 얻을 수 있다.

$$\hat{L}_k^E - w_{max} \le \hat{L}_k \le \hat{L}_k^E \tag{16}$$

식 (16)에서 k를 $k+1$로 대체한 후, 식 (16)에서 이를 빼면 다음의 관계를 얻을 수 있다.

$$\hat{L}_{k+1}^E - \hat{L}_k^E - w_{max} \le \hat{L}_{k+1} - \hat{L}_k \le \hat{L}_{k+1}^E - \hat{L}_k^E + w_{max} \tag{17}$$

다음으로, 식 (17)에 나타낸 $\hat{L}_{k+1} - \hat{L}_k$의 범위를 구하기 위해서 $\hat{L}_{k+1}^E - \hat{L}_k^E$의 범위를 구한다. 식 (15)에 정의된 $\hat{L}_k^E$에서, k를 $k+1$로 대체하면 다음 식을 얻을 수 있다.

$$L_{k+1}^E = \frac{\sum_{j=k+1}^{m} \hat{L}_j}{m-k} = \frac{(m-k+1)\hat{L}_k^E - \hat{L}_k}{m-k}$$

$$\Rightarrow \hat{L}_{k+1}^E - \hat{L}_k^E = \frac{\hat{L}_k^E - \hat{L}_k}{m-k} \tag{18}$$

식 (18)을 사용하여 식 (17)을 다시 쓰면 다음과 같다.

$$\frac{\hat{L}_k^E - \hat{L}_k}{m-k} - w_{max} \le \hat{L}_{k+1} - \hat{L}_k \le \frac{\hat{L}_k^E - \hat{L}_k}{m-k} + w_{max}$$

식 (15)와 SPLIT 의 정의된 동작에 의하여, $0 \le \hat{L}_k^E - \hat{L}_k \le w_{max}$ 은 당연히 만족한다. 결과적으로, 이를 위 식에 대입하면 다음의 식을 얻을 수 있다.

$$-w_{max} \le \hat{L}_{k+1} - \hat{L}_k \le \frac{w_{max}}{m-k} + w_{max} \le 2w_{max}$$

Step (b): ADJUST 는 $\hat{L}_{k+1} - \hat{L}_k < 0$인 경우, 반복적으로 G_k에서 G_{k+1}로 태스크를 이동 시킨다. 따라서 $\hat{L}_{k+1} - \hat{L}_k$가 취할 수 있는 하한 값은 0 이 된다. 또한, 태스크 한 개가 이동될 때 $\hat{L}_{k+1} - \hat{L}_k$가 취할 수 있는 최대 값은 $2w_{max}$ 이다. 따라서 $\hat{L}_{k+1} - \hat{L}_k$ 가 취할 수 있는 값의 상한 값은 SPLIT 의 상한 값과 동일하다. 결론적으로, 이 lemma 는 참이다. □

LEMMA 4.2. 동일 클러스터 내부의 임의의 태스크 그룹간의 스케일된 부하의 차이는 $2mw_{max}$보다 작거나 같다.

PROOF. ADJUST는 $\hat{L}_1 \le \hat{L}_2 \le \cdots \le \hat{L}_m$ 관계를 만족 시킨다. 그리고 같은 클러스터 내부에서 태스크 그룹간 가장 큰 스케일된 부하 차이는 항상 G_m과 G_1 사이에서 발생한다. 따라서 $\hat{L}_m - \hat{L}_1$은 다음의 식으로 나타낼 수 있다.

$$\hat{L}_m - \hat{L}_1 = (\hat{L}_m - \hat{L}_{m-1}) + (\hat{L}_{m-1} - \hat{L}_{m-2}) + \cdots + (\hat{L}_2 - \hat{L}_1)$$

$$\le 2w_{max} + 2w_{max} + \cdots + 2w_{max}$$

$$\le 2mw_{max}.$$

결론적으로, 이 lemma는 참이다. □

이어서, SVR−BASED LOAD BALANCING 알고리즘은 다음 lemma를 만족시킴을 보인다.

LEMMA 4.3. 동일 클러스터 내의 임의의 태스크 τ_i와 τ_j에 대하여, λ의 주기로 부하분산 될 때, 각 태스크의 SVR 값이 증가된 양에 대한 차이는 상수 값 이내로 제한되고, 이를 표현하면 다음과 같다.

$$-\lambda \cdot \left(\frac{1}{w_{min}} - \frac{1}{w_{min} + 2mw_{max}}\right) \leq \Delta\hat{v}_{i,j}(r) \leq \lambda \cdot \left(\frac{1}{w_{min}} - \frac{1}{w_{min} + 2mw_{max}}\right)$$

PROOF. 이를 증명하기 위하여, 먼저 $\Delta\hat{v}_i(r)$을 τ_i의 r번째 부하분산 주기인 $[(r-1)\lambda,\ r\lambda]$ 동안 발생한 SVR 증가 량으로 정의한다. SVR을 나타낸 정의에 의하여 이는 다음과 같이 표현될 수 있다.

$$\Delta\hat{v}_i(r) = \frac{1}{w(\tau_i)} \int_{(r-1)\lambda}^{r\lambda} R_i(t)\, dt \tag{19}$$

위 식의 적분으로 표현된 부분은 $R_i^{avg}(r)$ 값과 시간 $[(r-1)\lambda,\ r\lambda]$ 동안 수행된 τ_i 의 CPU 시간의 곱으로 나타낼 수 있다. 이때, 시간 $[(r-1)\lambda,\ r\lambda]$ 동안 수행된 τ_i의 CPU 시간은 스케일된 가중치에 비례하고, τ_i가 속한 실행 큐의 스케일된 부하에 반비례하므로, 식 (19)은 다음과 같이 나타낼 수 있다.

$$\Delta\hat{v}_i(r) = \frac{1}{w(\tau_i)} \cdot R_i^{avg}(r) \cdot \left(\lambda \cdot \frac{w(\tau_i)/R_i^{avg}(r)}{\hat{L}_k}\right) = \frac{\lambda}{\hat{L}_k}$$

위 식에서는 τ_i가 태스크 그룹 G_k에 속한 경우를 표현한다.

이를 이용하면, $\Delta\hat{v}_{i,j}(r)$ 다음과 같이 나타낼 수 있다.

$$\Delta\hat{v}_{i,j}(r) = \Delta\hat{v}_i(r) - \Delta\hat{v}_j(r) = \lambda \cdot \left(\frac{1}{\hat{L}_k} - \frac{1}{\hat{L}_l}\right)$$

위 식은 τ_i가 태스크 그룹 G_k에, 그리고 τ_j는 태스크 그룹 G_l에 속한 경우를 표현한 식이다.

이어서 다음 두 가지 경우를 각각 분리하여 생각한다: (a) $\Delta\hat{v}_{i,j}(r) \geq 0$ and (b) $\Delta\hat{v}_{i,j}(r) < 0$.

Case (a): $\Delta\hat{v}_{i,j}(r) \geq 0$ 인 경우.

LEMMA 4.2에 의하여 다음 식을 유도할 수 있다.

$$\Delta\hat{v}_{i,j}(r) \leq \lambda \cdot \left(\frac{1}{\hat{L}_k} - \frac{1}{\hat{L}_k + 2mw_{max}}\right) \leq \lambda \cdot \left(\frac{1}{\hat{L}_k} - \frac{1}{\hat{L}_k + 2mw_{max}}\right) \qquad (20)$$

식(20)의 최 우측 항은 $\hat{L}_k$ 값이 커짐에 따라 점차 감소한다. 따라서 $\hat{L}_k$이 최소 값을 가질 때, $\Delta\hat{v}_{i,j}(r)$ 는 최대 값을 갖게 된다. 이러한 성질을 이용하여 식 (20)을 다음과 같이 나타낼 수 있다.

$$0 \leq \Delta\hat{v}_{i,j}(r) \leq \lambda \cdot \left(\frac{1}{w_{min}} - \frac{1}{w_{min} + 2mw_{max}}\right)$$

Case (b): $\Delta\hat{v}_{i,j}(r) < 0$ 인 경우

Case (b)의 증명은 위 Case (a)와 비슷하게 이루어지며, 결론적으로 다음을 유도할 수 있다.

$$-\lambda \cdot \left(\frac{1}{w_{min}} - \frac{1}{w_{min} + 2mw_{max}}\right) \leq \Delta\hat{v}_{i,j}(r) < 0$$

Case (a)와 Case (b)를 조합하면 이 lemma가 참임을 알 수 있다. □

LEMMA 4.3을 사용하여 마지막으로, 본 연구에서 제안하는 스케줄링 기법이 동일 클러스터 내부의 임의의 두 태스크에 대하여, SVR 차이가 상수 값 이내로 제한됨을 증명한다.

THEOREM 1. 동일 클러스터 내의 임의의 두 태스크 τ_i와 τ_j에 대하여, $|\hat{v}_{i,j}(t)|$ 값은 상수 값 이내로 제한되며, 이를 식으로 나타내면 다음과 같다.

$$|\hat{v}_{i,j}(t)| \leq C\lambda \quad \text{where } C = \left(\frac{1}{w_{min}} - \frac{1}{w_{min} + 2mw_{max}}\right)$$

PROOF. $|\hat{v}_{i,j}(t)|$ 값이 가장 클 때는 매 부하분산 주기마다 발생한다. 따라서 본 증명 과정에서는 시간 $t = r\lambda$의 경우만 고려한다. 또한, 이를 증명하기 위하여 본 학위논문에서는 수학적 귀납법을 사용한다.

먼저 $t = 0$인 경우를 생각한다. 모든 태스크의 SVR 값은 t =0일 때 0이다. 따라서 이는 theorem을 만족한다.

다음으로 $t = r\lambda$일 때 theorem을 만족하면, $t = (r+1)\lambda$일 때도 만족함을 보인다. $t = (r+1)\lambda$일 때, 두 태스크의 SVR 값 차이는 다음과 같다.

$$\hat{v}_{i,j}(r+1)\lambda = \hat{v}_{i,j}(r\lambda) + \Delta\hat{v}_{i,j}(r)$$

먼저 $\hat{v}_{i,j}(r\lambda) < 0$인 경우를 생각한다. 이 경우, $\Delta\hat{v}_{i,j}(r)$ 값이 양의 값이고, $C\lambda$보다 작음을 뜻한다. 이는 LEMMA 4.3의 Case (a)를 통해서 쉽게 알 수 있다. 귀납법의 유도 가정(induction hypothesis)에 의하여 $\hat{v}_{i,j}(r\lambda) \geq -C\lambda$ 는 참이다. 그리고 $\hat{v}_{i,j}(r+1)\lambda$ 는 $\hat{v}_{i,j}(r\lambda)$ 보다 크다. 따라서 $-C\lambda \leq \hat{v}_{i,j}(r+1)\lambda \leq C\lambda$을 만족한다.

반대로, $\hat{v}_{i,j}(r\lambda) \geq 0$인 경우를 생각한다. 이는 $\Delta\hat{v}_{i,j}(r)$값이 음의 값이고, $-C\lambda$보다 큰 값임을 뜻한다. 이는 LEMMA 4.3의 Case (b)를 통해서 알 수 있다. $t = r\lambda$에서 이 theorem이 참임을 가정하였기 때문에, $\hat{v}_{i,j}(r\lambda) \leq C\lambda$ 는 참이다. $\hat{v}_{i,j}(r+1)\lambda$ 값은 $\hat{v}_{i,j}(r\lambda)$ 보다 작다. 따라서 $-C\lambda \leq \hat{v}_{i,j}(r+1)\lambda \leq C\lambda$을 만족한다.

결론적으로, $\hat{v}_{i,j}(r\lambda) < 0$ 인 경우와 $\hat{v}_{i,j}(r\lambda) > 0$ 인 경우 모두, $-C\lambda \leq \hat{v}_{i,j}(r+1)\lambda \leq C\lambda$을 만족함을 알 수 있다. 따라서 이 theorem 은 참이다. □

다음으로, 본 연구에서 제안하는 공정할당 스케줄링 기법의 시간 복잡도를 분석한다. 본 연구에서 대상으로 하는 클러스터는 앞서 설명한대로, n개의 태스크와 m개의 코어로 이루어진 클러스터이다. ALGORITHM 2에 기술된 SVR-BASED LOAD BALANCING 은 세 개의 서브루틴을 가지고 있다: SORT, SPLIT, ADJUST.

SORT 에서는 merge-sort를 수행하므로, 이의 시간 복잡도는 직관적으로 $O(n \log n)$임을 알 수 있다. SPLIT 이 수행되는 동안 각각의 태스크 그룹은 서로 다른 태스크들을 가지고 있다. 따라서 SPLIT 의 시간 복잡도는 $O(n)$ 이다.

다음으로 ADJUST 를 살펴본다. 독립적으로 CHECKANDMOVETASKS 함수가 호출될 때, 이는 $O(1)$의 시간 복잡도를 갖는다. 이 함수가 호출될

때, 최대 태스크 이주 발생 수는 $\frac{w_{max}}{w_{min}}$보다 작다. 그리고 전체적으로 이 함수는 $O(n)$번 불릴 수 있다. 따라서 ADJUST 는 $O(mn)$의 시간 복잡도를 갖는다. 전체적으로 보면, 본 연구에서 제안된 스케줄링 기법의 시간 복잡도는 $O(n\log n + n + mn) \approx O(nlog\ n)$이다.

제 5 장 실험 및 검증

제안된 각각의 스케줄링 기법의 실효성을 평가하기 위해서, 본 장은 수행한 실험에 대하여 설명한다. 실험 대상으로는 비대칭 멀티코어 아키텍처를 사용하는 상용 제품을 사용하였다. 본 장은 다음의 세 개의 절로 구성되어 있다. 1절은 비대칭 멀티코어 아키텍처의 코어 타입 선택 방식과 전체 코어 사용 방식을 나타내는 하드웨어적 환경을 기술한다. 또한, 이에 사용된 소프트웨어적 명세에 대하여 설명한다. 2절은 정량적으로, 제안된 기법의 실효성을 평가하기 위한 성능 지표들을 기술한다. 마지막으로, 3절에서는 구현된 기법들을 적용한 실험적 결과들을 검증한다.

제 1 절 실험 환경

본 연구의 목표는 비대칭 멀티코어 아키텍처가 나타내는 두 가지 모드에 최적화된 스케줄링 기법을 제안하는 것이다. 이를 위해서, 본 연구에서 제안된 스케줄링 기법들은 각 모드를 하드웨어적으로 지원하고 있는 상용 제품에 구현되었다. 구체적으로, 빅리틀 아키텍처의 CPU간 이주 모드에 적합한 스케줄링 기법은 Android 스마트폰 Galaxy S4위에 구현되었다. 또한, GTS 모드에 적합한 스케줄링 기법은 ARM사의 Versatile Express TC2 board에 구현되었다 [5][9]. 이들 대상 시스템에 대한 소프트웨어적 및 하드웨어적 자세한 구성 요소들은 표 4에 명시하였다.

표 4. 대상 시스템의 하드웨어 및 소프트웨어적 명세

구분			설명
비대칭 멀티코어 아키텍처용 저전력 스케줄링	HW	보드 명	Galaxy S4
		빅코어	Cortex-A15X4
		리틀코어	Cortex-A7X4
		메모리	2GB SDRAM
	SW	Android	Android version 4.1.1
		Linux	kernel version 3.4.0
비대칭 멀티코어 아키텍처용 공정할당 스케줄링	HW	보드 명	Versatile Express TC2
		빅코어	Cortex-A15X2
		리틀코어	Cortex-A7X3
		메모리	2GB DDR2
	SW	Android	Android version 4.4.2
		Linux	kernel version 3.10.54

제 2 절 실험 시나리오 및 성능 지표

본 절에서는 본 연구에서 수행했던 실험들이 어떤 테스트 시나리오와 workload를 사용했는지에 대하여 기술한다. 또한, 본 학위논문에서 제안된 스케줄링 기법들을 표 4에 나타낸 대상 시스템들에 구현했을 때, 최적화 달성 여부를 평가할 수 있는 정량화된 성능 지표를 정의한다. 본 연

구에서, 최적화 이슈는 (1) 빅리틀 아키텍처의 CPU간 이주 모드에서의 저전력 특성에 대한 최적화 달성 여부, 그리고 (2) GTS 모드에서의 공정할당성 최적화 달성 여부이다.

2.1 저전력 스케줄링 기법 최적화

본 연구에서 제안된 빅리틀 아키텍처의 CPU간 이주 모드용 스케줄링 기법이 최적화를 달성했는지 확인하기 위하여, 두 가지 종류의 실험을 수행하였다. 첫째, 벤치마크를 통한 저전력 특성을 체크하였다. 사용된 벤치마크들은 생성과 소멸을 반복한다. 따라서, 벤치마크들의 생성/소멸 비율을 조절하면서 여러 가지 서로 다른 실험 상황을 만들어 낸다. 둘째, 소프트웨어로 디코딩되는 비디오 플레이백을 사용하였다. 이 실험을 통하여, 실제 생활에서 사용되는 Android 응용프로그램들이 QoS를 해치지 않으면서 에너지 효율적으로 운용되는지 확인하였다.

본 연구에서 사용한 벤치마크는 SPEC CPU2006을 사용하였다. 이는 여러 종류의 CPU-intensive 벤치마크들로 이루어져 있으며, 각각의 벤치마크들은 모두 한 개의 태스크로 이루어져 있다. 이 중, 본 학위논문에서는 h264ref 와 gcc를 사용하였고, 이들에 대한 설명은 표 5에 나타내었다.[45][46]

이 벤치마크들의 특성은 실제 Android 응용프로그램과는 다르다. 이들은 일단 시작되면 실행을 마칠 때까지 Sleep 상태로 가거나, IO 동작에 의하여 블로킹되지 않는다. 따라서, 이들 벤치마크들은 시스템의 준비 큐(wait queue)로 진입하지 않고, 수행되는 동아 단지 부하분산 정책에 의하여 실행 큐(run-queue) 사이만 옮겨 다니게 된다.

표 5. 저전력 스케줄링 기법에 사용된 SPEC CPU2006 벤치마크

Test suites	Description
h.264ref	This test suite is a reference implementation of H.264/AVC video compression. It encodes input video data.
gcc	This test suite is based on gcc version 3.2. It generates code for AMD Opteron processor. It runs as a compiler with many of its optimization flgs enabled

하지만, 실제 Android 단말에서는, 어떤 한 응용프로그램이나 태스크가 특정 코어를 독점하는 현상을 막기 위하여, 준비 큐로 자주 태스크들이 옮겨가고, 다시 실행 큐로 할당이 된다. 본 연구에서 제안하는 스케줄링 기법은, 이러한 실제 Android 단말에서 사용되는 응용프로그램들을 운용함에 있어서 저전력효과를 나타내는데 목적이 있다. 따라서, 단순히 CPU-intensive 벤치마크들을 실험에 사용하면, 실제 Android 응용프로그램들의 동작 시나리오를 적절히 반영하지 못한다.

따라서, 본 연구에서 제안된 스케줄링 기법을 테스트하기 위하여, 그림 18에 보이는 것처럼 실험 시나리오를 구성하였다. 실험 시나리오에서는 SPEC CPU 2006에서 제공하는 h264ref와 gcc 두 개의 벤치마크를 사용하였다.

h264ref의 encoding 프레임 수는 2로 설정하였고, gcc가 컴파일하는

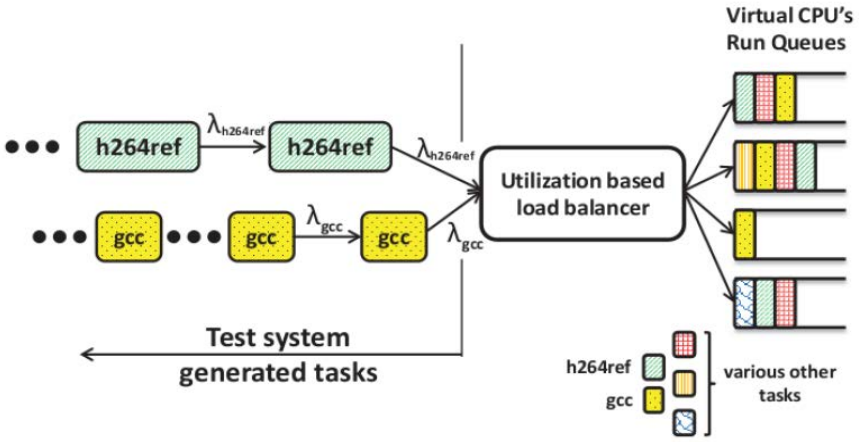


그림 18. h264ref 와 gcc 벤치마크를 사용한 태스크의 생성/소멸 반복

데 사용하는 파일의 크기는 3.4KB로 하였다. h264ref 태스크들은 그림의 사용률 기반 부하분산기(utilization based load balancer) 앞에 고정된 도착률(1/3 arrivals/second)로 도착한다.

일단 h264ref 태스크가 부하분산기에 의하여 가상 CPU에 할당이 되면, 계속 수행되다가 시스템에서 빠져나간다. 하지만 동일한 태스크는 다시 생성되어 시스템에 진입하게 된다. 따라서, 이러한 실험 환경은 태스크가 가상 CPU에서 실행을 멈추고 준비 큐로 진입 후, 다시 실행 큐 할당을 기다리는 상황과 동일한 환경을 제공한다.

본 실험에서는 $\lambda_{h264ref}$ 를 h264ref의 도착률, gcc의 도착률은 λ_{gcc} 로 표현하였다. 여기에서 도착률이란 새로운 h264ref와 gcc 태스크가 생성되어 부하분산기 앞에 도착하는 비율을 나타낸다. 즉, 다시 말하면 1초동안 몇 개의 태스크가 생성되는지를 나타낸다. 이 두 도착률은 실험을 수행하는 동안 변하지 않고 고정된다. 본 연구에서는 $\lambda_{h264ref}$ 와 달리 λ_{gcc} 는 여러 가지 값을 취하여 각각 실험하였다. 즉, [20 arrivals/second, 80

arrivals/second] 사이의 숫자를 취하게 하였으며, 그 간격은 10 arrivals/second으로 하였다.

실험에서 측정 시, 각 가상 CPU당 평균 gcc의 서비스률 μ_{gcc}는 14.3 tasks/second 혹은, 약 70ms/task 였다. 따라서, 이는 시스템이 57.2 gcc tasks/seconds를 운용할 수 있다는 뜻이다. 도착률이 50 arrivals/second 혹은, 이보다 작을 때, 태스크들은 wait → run → wait loop 루틴을 반복하고, 이때 어떤 고정된 값의 wait 시간을 갖는다. 도착률이 60 arrivals/second 혹은 이보다 큰 값을 가질 때는, 앞서 생성된 태스크가 그 실행을 마치기 전에 새로운 태스크가 생성되어 시스템으로 유입됨을 알 수 있었다.

이렇게 태스크의 도착률을 변화시킴으로써, 태스크가 서로 다른 wait 시간을 가지면서, 마치 시스템의 준비 큐에 머물게 하는 효과를 만들었다. 또한, 이는 실제 Android의 응용프로그램들이 준비 큐에 머물다가 실행 큐를 선택해서 진입하는 시나리오를 재연하였다. 이 실험 방법은 벤치마크에 인위적인 wait 상태를 부과하되, SPEC CPU2006 벤치마크를 수정하지 않고 사용할 수 있게 하였다.

SPEC CPU2006 벤치마크를 수행한 후 저전력 성능 지표로서, 소비된 에너지(mAh)를 측정하였다. 이는 gcc 태스크간 도착률 λ_{gcc} 을 [20 arrivals/second, 80 arrivals/second] 사이의 값을 갖게 하면서 측정하였다. 이와 동시에, 각 λgcc 마다 gcc가 수행을 완료한 시간을 측정하였다. 이는 에너지소비와 성능의 관계에 있어서, 제안된 스케줄링 기법이 얼마 만큼의 성능을 감소시키면서 에너지소비를 줄이는지 확인하기 위해서이다.

두 번째 실험으로, 하드웨어 디코더를 쓰지 않고, 100% 소프트웨어로 디코딩되는 Android MX player를 실험 대상으로 사용하였다. MX player

는 21개의 태스크들로 이루어진 Android 응용프로그램이다. 이들 21개의 태스크들은 가상 CPU에서 수행된 후, Sleep과 같은 블로킹 상태를 만나서 시스템의 준비 큐에 옮겨진다. 또한 이들은 부하분산 정책에 의하여 실행 큐들 사이를 옮겨 다닌다.

이와 같은 전형적인 Android 응용프로그램의 특성을 가진 MX player를 본 연구에서 활용함으로써, 제안된 기법의 저전력 최적화 달성 여부를 판단할 수 있다. 실험 대상으로 사용한 디코딩용 파일은 h.264 codec으로 압축된 1920X1080 full HD 비디오 clip이다. 본 실험에서 사용된 이 비디오 clip은 31초 동안 디코딩하여 재생시키는 콘텐츠이다.

MX player를 수행한 후 저전력 성능 지표로서, 소비된 에너지(mAh)를 측정하였다. 이는 31초동안 비디오 clip을 재상 하면서 소비된 에너지이다. 이와 동시에, FPS(Frame per Second)를 1초마다 측정하고, 평균값을 측정하였다. 이는 에너지소비와 QoS의 관계에 있어서, 제안된 스케줄링 기법이 얼마 만큼 Android 응용프로그램의 QoS 특성을 해치면서 에너지소비를 줄이는지 확인하기 위해서이다.

2.2 공정할당 스케줄링 기법 최적화

본 학위논문에서 제안하는 공정할당 스케줄링 기법의 최적화 달성 여부를 확인하기 위해서, 다음의 세 가지 실험을 수행하였다: (1) 태스크 간 $\hat{v}_{max}(t)$를 측정, (2) 동일한 태스크를 여러 개 수행시킨 후 각 태스크들의 완료시간 편차를 측정, (3) 제안된 기법으로 인하여 발생하는 런타임 오버헤드 측정. 열거한 세 가지 실험에서 부하분산 주기 λ는 기존의 CFS와 동일하게 1초로 하였다.

첫째, 본 연구에서 제안한 공정할당 스케줄링 기법을 적용하였을 때와 하지 않았을 때의 $\hat{v}_{max}(t)$를 측정하였다. 제안된 기법의 실효성을 검증하기 위해서, 시스템 운용시간 증가에 따른 $\hat{v}_{max}(t)$ 변화 값을 비교 하였다. 4.1절에서 먼저 기술하였듯이, 이상적인 공정할당 스케줄러는 $\hat{v}_{max}(t)$를 0으로 유지시킨다. 따라서, 본 실험에서는 시간이 증가하여도 이 값이 발산하지 않고 상수 값 이내로 제한됨을 검증하고자 한다.

시간에 따른 $\hat{v}_{max}(t)$ 값 변화를 측정하기 위해서, 한 개의 태스크로 구성된 응용프로그램과 여러 개의 태스크로 구성된 응용프로그램을 사용하였다. 이를 위해서 SPEC CPU2006 과 PARSEC 벤치마크를 각각을 나타내는 부하로 사용하였다 [25][26][45][46]. SPEC CPU2006 벤치마크에서는 bzip2, bwaves, mcf 등을 사용하고, PARSEC 벤치마크 중 swaptions, blacksholes, ferret 등을 실험에 사용하였다. 그리고 이들에 대한 설명을 표 6, 7에 나타내었다. SPEC CPU2006 벤치마크를 사용시

표 6. 공정할당 스케줄링 기법에 사용된 SPEC CPU2006 벤치마크

Test suites	Description
bzip2	This test suite is based on an open-source file compression program named bzip2. Each input is compressed and decompressed.
bwaves	This test suite is a floating point benchmark. It numerically simulates blast waves in three dimensional transonic transient laminar viscous flow.
mcf	This test suite is a program used for single-depot vehicle scheduling in public mass transportation.

에는 동일한 16개의 벤치마크를 시스템에 부가하였고, PARSEC 벤치마크에서는 여러 개의 태스크로 이루어진 한 개의 벤치마크를 사용하였다. 본 실험에서는 $\hat{v}_{max}(t)$ 값을 초단위로 측정하여 성능 지표로 사용하였다.

둘째, $\hat{v}_{max}(t)$이 발산하지 않고 상수 값 이내로 제한되는 것이 공정할당에 어떤 영향을 주는지, 보다 직관적으로 확인하였다. 이를 위하여, 본 실험에서는 동일한 workload를 여러 개 복사해서 시스템에 부가하였다. 해당 workload는 한 개의 태스크로 구성된 응용프로그램이며, 이는 CPU-intensive한 bubble-sort를 수행하고 정해진 시간 동안 sleep하는 간단한 동작을 반복적으로 수행한다.

표 7. 공정할당 스케줄링 기법에 사용된 PARSEC 벤치마크

Test suites	Description
blackscholes	This test suite runs a computational finance application which calculates the prices for a portfolio via the BlackScholes partial differential equation (PDE). It is the simplest of all PARSEC workload.
swaptions	This test suite runs a computational finance application which employs a Monte Carlos simulation to analyze non Markovian models.
ferret	This test suite is a This application is based on the Ferret toolkit which is used for content-based similarity search of feature-rich data such as audio, images, video, 3D shapesprogram used for single-depot vehicle scheduling in public mass transportation.

본 실험에서 CPU-intensive하고 한 개의 태스크로 구성된 SPEC CPU2006을 사용하지 않고, 앞서 설명한 인위적인 응용프로그램을 사용한 이유는 다음과 같다. 저전력 스케줄링 기법 최적화 달성 여부를 확인하는 실험에서 설명되었듯이, SPEC CPU2006 벤치마크의 태스크들은 일반적인 응용프로그램과 달리 시스템의 준비 큐로 이동하지 않고, 코어와 실행 큐에만 존재한다. 따라서 본 실험에서도 실제상황에 가까운 workload를 나타내기 위해서 실행 큐와 준비 큐를 옮겨 다니는 부하를 사용하였다.

성능 지표로서는 각 응용프로그램들의 완료시간을 사용하였다. 즉, 이상적인 공정할당 스케줄링 기법을 사용할 경우, 각각의 동일 태스크들은 같은 시점에 완료되어야 한다. 따라서, 본 실험에서는, 각 태스크들이 나타내는 완료시간의 표준편차, 완료시간의 최대/최소 값의 차이 등을 측정하였다.

셋째, 제안된 공정할당 스케줄링 기법이 런타임에 나타내는 오버헤드를 측정하였다. 이를 위해서, 앞서 사용했던 SPEC CPU2006 과 PARSEC 벤치마크를 사용하였다. 런타임 오버헤드를 파악하기 위하여, 제안된 기법이 적용되었을 경우, 벤치마크 완료시간이 적용되기 전보다 얼마나 늘어났는지를 측정하였다.

제 3 절 실험적 검증 결과

본 절에서는 빅리틀 아키텍처의 (1) CPU간 이주 모드용 저전력 스케줄링 기법과 (2) GTS 모드용 공정할당 스케줄링 기법에 대한 최적화 달성 여부를 실험적 검증 내용을 통하여 기술한다. 3.1절에서는 저전력 최적화 특성을 벤치마크 수행 시 감소된 에너지 사용으로써 확인한다. 또한,

Android 응용프로그램을 수행 시 에너지 소비가 감소하는 지를 살펴본다. 마지막으로, 제안된 기법으로 인한 런타임 오버헤드를 확인하기 위하여, 벤치마크 수행시간을 비교하고, Android 응용프로그램의 QoS 특성 저하 여부를 확인한다.

3.2 절에서는 공정할당 스케줄링 기법의 실험적 검증 검증 결과를 태스크 간 최대 SVR 차이가 상수 값 이내로 제한됨을 통하여 확인한다. 또한, 이러한 결과가 여러 개의 동일한 태스크들 수행 시, 그들의 완료 시간 편차가 감소하는 결과로 나타남을 검증한다. 마지막으로, 제안한 기법들로 인하여 발생하는 런타임 오버헤드에 대하여 설명한다.

3.1 저전력 스케줄링 기법의 실험 결과

■ **벤치마크 테스트**

그림 19는 gcc 벤치마크의 도착률 λ_{gcc}를 [20 arrivals/second, 80 arrivals/second] 사이의 값을 취하면서 측정한 에너지 소비(mWh) 결과를 나타낸다. 그림에서, Legacy의 의미는 본래의 CFS에서의 결과를 나타내고, $E_A(U)$와 $E_B(U)$는 각각 3장에서 설명한 사용률 기반 추정기와 수행이력을 반영한 사용률 추정기를 사용했을 때의 결과를 나타낸다. 그림에 나타낸 세 개의 직선은, λ_{gcc}를 변화시키며 기록한 실험 결과를 Linear regression을 사용하여 나타낸 결과이다.

그림에서 알 수 있듯이, $E_A(U)$ 혹은 $E_B(U)$ 추정기를 사용하는, 본 연구에서 제안한 스케줄링 기법이 에너지 소비를 감소시켰다. Linear regression이 나타내는 라인을 보았을 때, λ_{gcc} 값이 커질 때 즉, 시스템에 부하가 더 많아질 때 소비된 에너지 차이는 더 커짐을 알 수 있다.

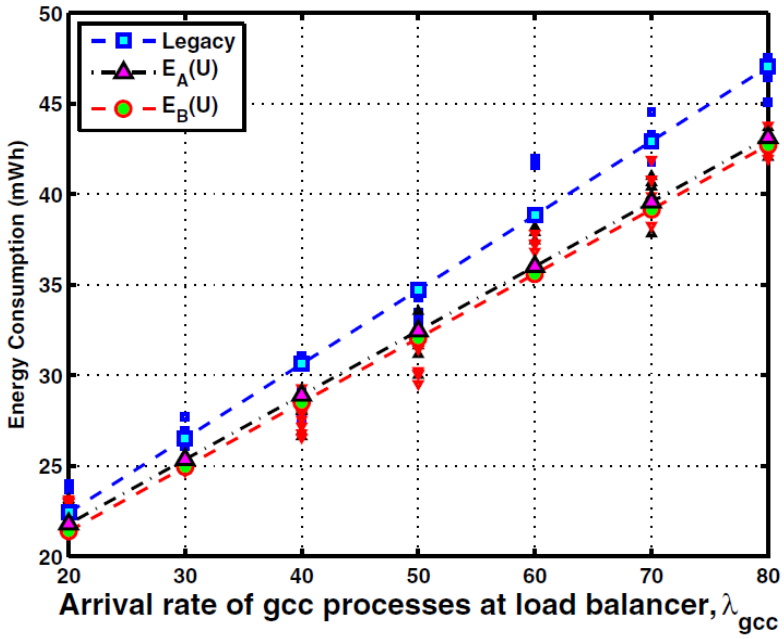


그림 19. gcc 도착률에 따른 에너지 소비 비교

표 8은 그림 19에 나타낸 λ_{gcc} 값들 중, 20, 30, 60 arrivals/second의 도착률일 때 소비된 에너지를 구체적인 값으로 나타내고 있다. 또한, 표에서는 제안된 스케줄링 기법에 의하여 발생한 런타임 오버헤드를 나타내고 있다. 표에서 나타낸 Legacy, $E_A(U)$, $E_B(U)$의 의미는 앞서 그림 19에서 설명한 바와 동일하다. $\Delta_A\%$ 와 $\Delta_B\%$는 각각 본래의 CFS와 $E_A(U)$ 및 $E_B(U)$가 나타내는 값을 비교했을 때, 측정된 값의 변화율을 나타낸다.

표에서 알 수 있듯이, 소비 전류, 전력, 에너지 모든 측면에서 제안된 스케줄링 기법이 더 작은 값을 나타내었다. 또한, 사용률 기반 추정기를 사용했을 때보다 수행이력을 반영한 사용률 추정기를 사용했을 때, 저전

력 효과가 더 큼을 알 수 있었다. 본 학위 논문에서 제안한 두 추정기의 차이점은 다음과 같다.

수행이력을 반영한 사용률 추정기는, 어떤 태스크가 준비 큐에서 빠져

표 8. gcc 도착률에 따른 에너지 소비 및 런타임 오버헤드

λ_{gcc}	구분	Legacy	$E_A(U)$	Δ_A%	$E_B(U)$	Δ_B%
20	Time (s)	36.40	36.44	0.12%	36.67	0.75%
	Current (mA)	591.87	577.74		570.06	
	Power (mW)	2359.81	2303.63		2272.79	
	Energy (mWh)	23.86	23.31	−2.29%	23.15	−2.96%
30	Time (s)	38.65	39.66	2.62%	39.84	3.09%
	Current (mA)	625.26	573.90		563.04	
	Power (mW)	2493.04	2288.24		2244.89	
	Energy (mWh)	26.77	25.21	−5.83%	24.84	−7.18%
60	Time (s)	50.88	52.84	3.84%	52.54	3.25%
	Current (mA)	743.24	636.81		638.29	
	Power (mW)	2963.27	2539.00		2544.80	
	Energy (mWh)	41.88	37.26	−11.04%	37.13	−11.35%

나와 실행 큐로 진입할 때, 그 태스크가 가상 CPU 사용률을 변화 시키는 정도를 예측한 후 실행 큐를 결정한다. 하지만 사용률 기반 추정기는, 태스크가 어떠한 실행 큐로 진입하더라도, 모든 가상 CPU들의 사용률은 진입 전과 동일하다는 가정하에 실행 큐를 결정한다. 즉, 현재 시점의 가상 CPU 사용률만 보고 판단한다. 따라서, 제안한 스케줄링 기법의 사용률 예측 방법이 실효성이 있다는 것을 검증하고 있다.

표에 나타낸 Time (s)은 gcc 벤치마크의 완료시간 초단위로 나타낸 결과이다. Legacy와 두 가지 제안된 추정기를 사용했을 때를 비교하면, 0.12% ~ 3.84% 수행시간 증가를 나타내었다. 즉 이는 런타임 오버헤드를 나타내며, 시스템의 부하가 커지더라도, 아주 작은 폭으로 증가함을 알 수 있었다.

■ Android 응용프로그램 테스트

벤치마크를 활용한 검증에 부가하여, 실제 Android 응용프로그램에서도 제안된 저전력 스케줄링 기법이 실효성이 있는지 확인하였다. 이를 위하여 소프트웨어로 디코딩되는 Android MX player를 사용하였다.

표 9는 이의 결과를 나타내고 있다. 이 표에 나타낸 Legacy, $E_A(U)$, $E_B(U)$의 의미는 벤치마크 테스트 결과를 설명할 때 기술한 바와 같다. 실험에서, 완전한 frame rate를 나타내기 까지, Legacy, $E_A(U)$, $E_B(U)$ 모두 2초간의 준비 시간이 필요했다. 또한, MX player가 실행을 마치는 시점의 마지막 2초 역시, 세가지 실험 환경 모두 완전한 frame rate를 나타내지 못했다. 따라서 본 실험에서, 시작 시점과 마지막 시점의 총 4초는 QoS 지표인 평균 FPS 계산시 제외시켰다. 하지만 다른 측정 결과에는 해당 4초동안의 데이터를 반영하였다.

표 9. 소프트웨어로 디코딩된 MX player 실험 결과

	Legacy-CPU	$E_A(U)$	$E_B(U)$
Time (s)	31.53	31.39	31.43
Current (mA)	690.78	650.46	644.28
Power (mW)	2754.34	2593.50	2568.94
Energy (mWh)	24.12	22.61	22.43
Energy Δ%	-	-6.68%	-7.53 %
Mean FPS	30.0	29.964	30
σ_{FPS}	0	0.429	0.385

표 9에서 알 수 있듯이, 소비 전류, 전력, 에너지 모든 측면에서 제안된 스케줄링 기법이 더 작은 값을 나타내었다. 특히, 수행이력을 반영한 사용률 추정기($E_B(U)$로 표시)를 사용했을 때, 앞서 기술한 벤치마크 테스트 결과와 마찬가지로, 더 큰 에너지 감소를 나타내었다. 이러한 결과로 인하여, 실제 Android 응용프로그램에서도 제안된 스케줄링 기법의 사용률 예측 방법이 실효성이 있다는 것을 검증하고 있다

제안된 스케줄링 기법이 침해할 수 있는 QoS 특성을 파악하기 위하여, 평균 FPS를 측정하였다. 표에서 알 수 있듯이, $E_B(U)$로 표시된, 수행이력을 반영한 사용률 추정기를 사용했을 때는 QoS 특성 침해가 없었다. 또한, $E_A(U)$로 표시한 사용률 기반 추정기 사용시에도, 0.1%의 사용자가

체감할 수 없는 정도의 작은 성능 손실이 관찰되었다.

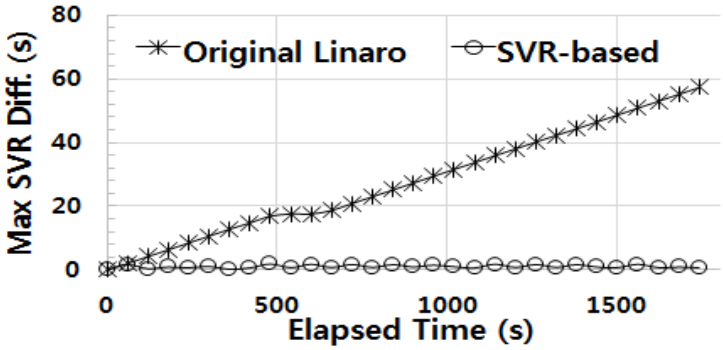
3.2 공정할당 스케줄링 기법의 실험 결과

■ **태스크간 SVR 차이 측정**

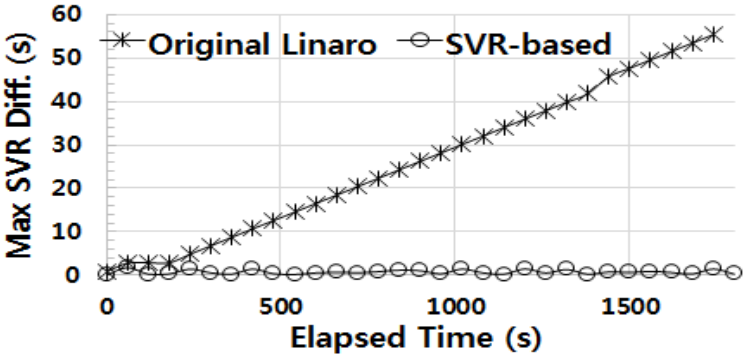
제안된 공정할당 스케줄링 기법이 빅리틀 아키텍처의 GTS 모드에서 최적화를 달성하는지 확인하기 위해서, 먼저 태스크 간의 $\hat{v}_{max}(t)$ 값을 본 학위논문에서 제안한 기법을 적용했을 때와 적용하지 않았을 때를 비교하였다.

그림 20는 한 개의 태스크로 구성된 bzip2를 16개 복사해서 시스템에 부가한 결과이고, 그림 21은 swaptions 한 개를 16개의 태스크로 분산시켜 수행했을 때 결과를 나타낸다. 본 학위논문에서 제안하는 스케줄링 기법의 핵심은 두 개의 클러스터 내부에서 각각 $\hat{v}_{max}(t)$를 상수 값 이내로 제한시키는 것이다. 따라서 본 실험에서는 각 클러스터별 $\hat{v}_{max}(t)$를 측정한다. 이어 클러스터 구분 없이 시스템 전체적으로 태스크 간 $\hat{v}_{max}(t)$를 측정하여, 클러스터 별로 $\hat{v}_{max}(t)$값을 상수 값 이내로 제한함이 시스템 전체적으로 어떤 영향을 미치는지 확인하였다.

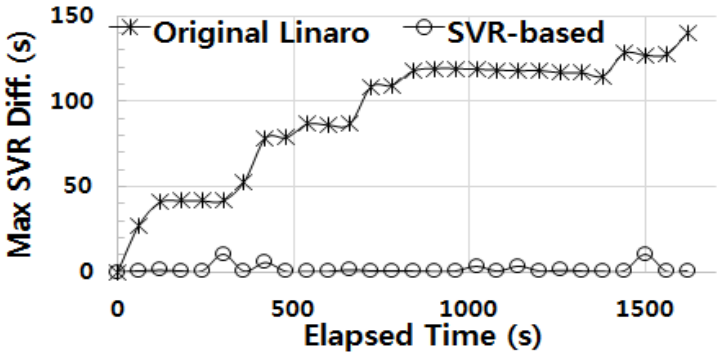
그림 20과 그림 21의 (a)는 빅클러스터에서 $\hat{v}_{max}(t)$를 측정한 결과이고, 그림 20과 그림 21의 (b)는 리틀클러스터에서 $\hat{v}_{max}(t)$를 측정한 결과이다. 그림에서 알 수 있듯이, 클러스터 타입과 벤치마크 종류에 관계없이 제안된 스케줄링 기법은 $\hat{v}_{max}(t)$ 값을 상수 값 이내로 제한시키고 있다. 하지만, 본래의 Linaro 스케줄링 프레임웍의 결과에서는 $\hat{v}_{max}(t)$ 값이 실험 시간이 증가함에 따라 발산함을 알 수 있다.



(a)

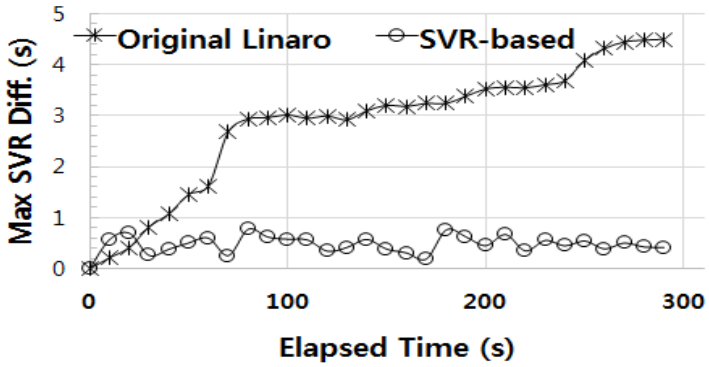


(b)

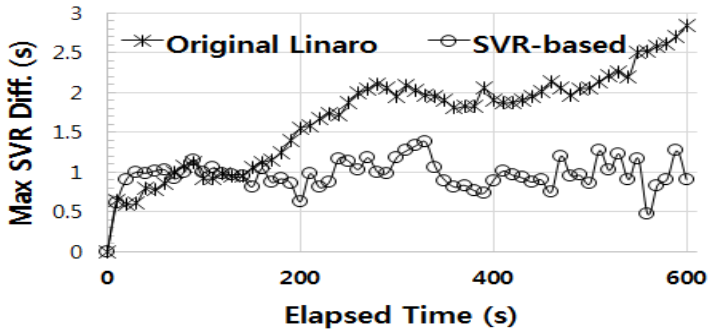


(c)

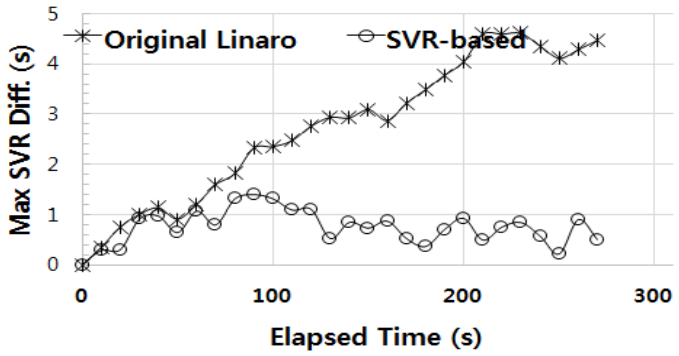
그림 20. SPEC CPU2006 (bzip2)를 사용한 최대 SVR 차이



(a)



(b)



(c)

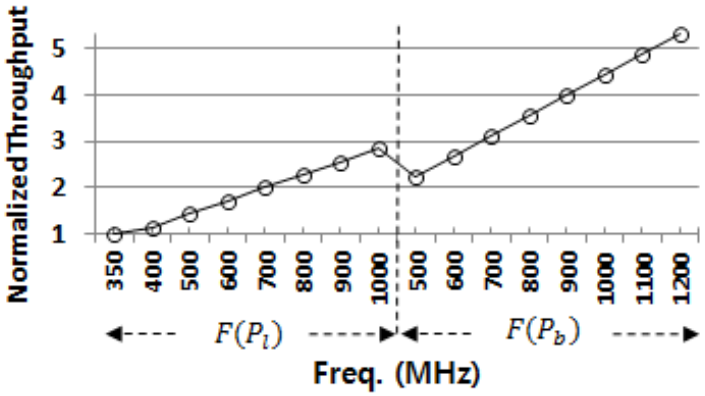
그림 21. PARSEC (swaptions)를 사용한 최대 SVR 차이

그림 20와 그림 21의 (c)는 클러스터를 구분하지 않고 시스템 전체의 $\hat{v}_{max}(t)$ 값을 측정한 결과이다. 본래의 Linaro 스케줄링 프레임웍을 적용할 경우, 앞서 설명한 클러스터별 결과와 동일하게 시간이 지남에 따라 계속 $\hat{v}_{max}(t)$ 값이 발산함을 알 수 있다. 반면에, 본 연구에서 제안한 스케줄링 기법을 적용 시, 동일한 16개의 벤치마크 테스트에서는 10.3초, 16개의 태스크로 이루어진 한 개의 벤치마크 테스트에서는 1.4초로 제한됨을 알 수 있었다.

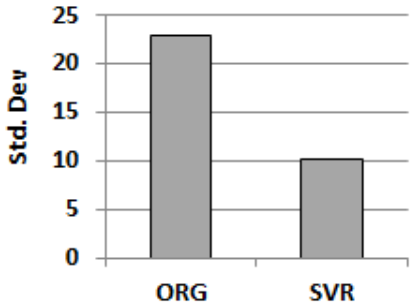
■ **동일 workload 테스트**

다음으로, 유한한 양의 CPU-intensive한 일을 수행하고, 한 개의 태스크로 이루어진 workload를 여러 개 복사하였다. 이어서, 이들을 시스템에 부가한 후, 이들의 완료 시간 편차를 측정하였다. 그림 22의 (a)는 이 workload의 각 f'에 따른 성능을 나타내고 있다. 이를 위하여 offline에서, 사용 가능한 f' 값들을 고정 시킨 후 각각의 성능을 측정하였다. 이때 $f' = f'_{min}$ 인 경우의 측정된 성능을 baseline으로 설정하고, 나머지 각각의 f'에서 측정된 성능들을 baseline에 대한 배수로 설정하였다. 그림에서 f'_{min}는 350 MHz를 나타낸다.

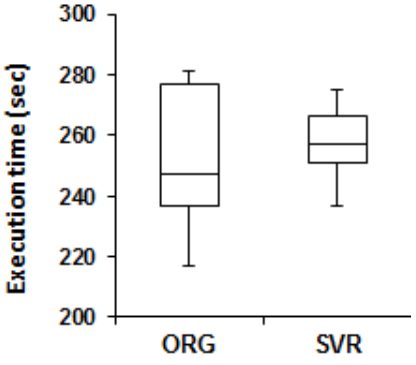
그림 22에서 ORG는 본래의 Linaro 스케줄링 프레임웍의 결과를 나타내고, SVR은 본 연구에서 제안된 스케줄링 기법의 결과를 나타낸다. 그림의 (b)는 동일한 16개 workload의 완료시간 표준편차를 나타내고 있다. 그림에서 알 수 있듯이, 제안된 스케줄링 기법이 원래의 Linaro 스케줄링 프레임웍보다 표준편차가 56% 작았다. 그림의 (c)는 동일 workload들 각각의 완료시간 분포를 나타내고 있다. 그림에서 보이듯이, 완료시간 최대편차 역시 제안된 스케줄링 기법이 49.6% 더 작은 값을 나타내었다.



(a)



(b)



(c)

그림 22. 16개 동일 workload의 완료시간 편차

벤치마크 및 동일 workload 를 사용한 실험들을 통하여, 제안된 공정할당 스케줄링 기법이 비대칭 멀티코어 아키텍처에서 향상된 공정할당성을 나타냄을 쉽게 알 수 있다.

■ **런타임 오버헤드 측정**

마지막으로 공정할당 스케줄링 프레임웍이 야기하는 오버헤드를 설명한다. 제안된 기법으로 인한 오버헤드는 기존의 Linaro 스케줄링 프레임웍에 추가로 탑재되거나 수정된 소프트웨어 모듈로 인한 지연시간으로 정의한다. 이러한 지연시간의 원인이 될 수 있는 점들을 다음과 같이 설명할 수 있다.

본 연구에서는, 기존의 virtual runtime 기반의 per-core 스케줄러를 SVR(scaled virtual runtime) 기반으로 변경하였다. 그리고 태스크 별 SVR 값을 구하기 위하여 SVR calculator를 추가하였다. 이에 매 스케줄링 tick마다 발생하는 SVR calculator의 OLS regression 동작이 지연시간을 초래할 수 있다.

태스크간 $\hat{v}_{max}(t)$를 상수 값 이내로 제한하기 위하여, 기존의 가중치 기반 부하분산 정책을 SVR 기반 부하분산으로 변경하였다. 기존의 가중치 기반 부하분산 정책은 실행 큐간 부하의 차이가 허락된 값 이내로 들어오는지 체크한 후 태스크를 옮긴다. 이에 반하여 제안된 스케줄링 기법은 SORT, SPLIT, ADJUST 등의 서브루틴을 수행하면서, merge-sort 및 태스크 그룹핑 작업을 수행한다. 이렇게 변경된 부하분산 방식이 지연요소로 나타날 수 있다.

이러한 지연시간을 발생시킬 수 있는 점들을 확인하기 위하여 전체론적인 관점에서 런타임 오버헤드를 측정하였다. 다시 말하면, 런타임 오버헤드로 인하여 태스크들이 지연되어 완료하는 점을 활용한다.

이를 위하여, 한 개의 태스크로 이루어진 bzip2, bwaves, mcf 등의 SPEC CPU2006 벤치마크와 여러 개의 태스크로 이루어진 swaptions, blacksholes, ferret 등의 PARSEC 벤치마크를 사용하였다. $\hat{v}_{max}(t)$ 측정 시 사용한 방법과 동일하게, SPEC CPU2006에 속하는 벤치마크들은 동일하게 16개를 복사하여 시스템에 부가하였고 PARSEC에 속하는 벤

치마크들은 16개의 태스크를 한 개의 응용프로그램으로 생성 후 시스템에 부가하였다.

표 10은 벤치마크 수행 결과를 제안된 기법을 적용했을 때와 적용하지 않았을 때를 비교한 것이다. SPEC CPU2006에 속하는 벤치마크들은 서

표 10. 벤치마크로 측정한 런타임 오버헤드

Benchmark		**Elapsed time(s)**	**Std. Dev.**	**Overhead (%)**
bzip2	ORG	889	212	-
	SVR	888	204	-0.1
bwaves	ORG	279	69	-
	SVR	282	55	0.92
mcf	ORG	488	56	-
	SVR	491	56	0.77
swaptions	ORG	68		-
	SVR	69		0.82
blackscholes	ORG	667		-
	SVR	655		-1.7
ferret	ORG	75		-
	SVR	76		1.34

로 동일한 태스크를 부가하였기에 그들간의 완료 시간 표준 편차를 측정하였고, PARSEC에 속하는 벤치마크들은 서로 다른 동작을 수행하는 16개의 태스크들이기에 이들의 표준 편차는 기입하지 않았다. 표에서 볼 수 있듯이 평균 런타임 오버헤드는 0.34%에 불과하였다. 또한, 한 개의 태스크로 이루어진 workload와 여러 개의 태스크로 이루어진 workload에서, 런타임 오버헤드 차이가 나타나지 않았다.

이렇게 추가되거나 변경된 소프트웨어 모듈로 인한 런타임 오버헤드가 작은 이유는 다음과 같다. OLS regression 수행 시, 20 tick 동안의 IPT(Instruction per tick) 데이터를 regression한다. 이때, 20 tick에 해당하는 데이터를 매번 측정하지 않고, 매 tick마다 최근 1개의 IPT 데이터를 추가하고 가장 오래된 데이터는 버린다. 이러한 점이 큰 오버헤드로 작용하지 않았다. 그리고 모든 태스크의 20 tick 동안의 IPT 데이터를 저장해야 하는, 메모리 사용측면에서의 오버헤드가 있을 수 있다. 본 연구에서는 이에 필요한 데이터 size를 최소화 하였고, 실제 size는 전체적으로 수 KB에 불과하였다.

다음으로 부하분산 측면에서 살펴본다. 기존 Linaro 스케줄링 프레임웍 대비, 제안된 스케줄링 기법은 merge-sort 및 태스크 그루핑 등이 추가되었다. 이는 실제 태스크를 이주시키기 전에 취하는 단순 연산 작업이며, 전체론적인 관점에서 보면 차지하는 시간은 1% 보다 작은 값이다.

제 6 장 결 론

본 논문은 임베디드 시스템에서 그 사용이 늘어가고 있는, 비대칭 멀티코어 아키텍처의 하드웨어적 동작방식에 따른 최적화된 태스크 스케줄링 기법을 제안하였다. 구체적으로, 비대칭 멀티코어 아키텍처의 코어 타입 선택 방식과 전체 코어 사용 방식에 사용되는 스케줄링 기법의 문제점을 지적하고, 각각에 대한 최적화 방법을 제시하였다.

첫째, 코어 타입 선택 방식에 최적화된 스케줄링 기법을 제안하였다. 이를 위하여 먼저, Linux kernel이 비대칭 멀티코어 아키텍처를 위하여 제공하는 DVFS 정책을 분석하였다. 분석된 내용을 토대로, 코어의 사용률에 의해서 변경되는 동작 주파수와 코어 타입 선택 과정을 하나의 상태도로 간략하게 표현하였다. 이 상태도를 기반으로, 태스크가 코어에 할당될 때 사용률을 추정하는 메커니즘을 Linux kernel의 부하분산 정책에 반영하였다. 이 메커니즘으로 인하여, 현재 운용되는 코어 타입을 직접적으로 알지 않아도, 에너지 효율적으로 태스크를 코어에 할당 할 수 있는 능력을 Linux kernel이 갖추게 되었다.

둘째, 전체 코어 사용 방식을 위한 공정할당 스케줄링 기법을 제안하였다. 이를 위하여 먼저, Linux kernel의 CFS를 확장하여 전체 코어 사용 방식에 맞게 구성한 Linaro 스케줄링 프레임웍을 분석하였다. 분석한 결과, 동일한 태스크들도 코어의 동작 주파수 혹은 코어 타입에 의존적으로 수행한 정도가 다르게 나타날 수 있으나, 이러한 점이 CFS에 반영되지 않음을 알 수 있었다. CFS는 태스크들의 상대적 진척도를 virtual runtime을 통하여 나타낸다. 따라서 본 연구에서는 성능 비대칭성을 반

영한 SVR(scaled virtual runtime)을 정의하고, 이를 CFS의 per-core 스케줄러에 반영하였다. 또한, 기존의 CFS는 모든 태스크들의 상대적 진척도를 비슷하게 유지하기 위하여 가중치를 기반으로 부하분산을 수행함을 분석을 통하여 알았다. 본 연구에서는 이를 개선하여 부하분산 시 모든 태스크들의 SVR(scaled virtual runtime)이 비슷하게 유지하는 알고리즘을 제안하였다. 구체적으로, 동일 클러스터 내부의 태스크들간 최대 SVR 값 차이가 작은 상수 값 이내로 제한되게 하였다.

비대칭 멀티코어 아키텍처의 두 가지 하드웨어적 동작 방식에, 제안된 스케줄링 기법들이 효용성이 있다는 것을 검증하였다. 이를 위해서 본 학위논문에서는 ARM사의 빅리틀 아키텍처를 대상시스템으로 하였다. 각각의 기법들을 빅리틀 아키텍처를 기반으로 하는 실제 상용제품들에 구현하였다. 구체적으로, 코어 타입 선택 방식에 최적화된 스케줄링 기법은 Galaxy S4 Android 스마트폰에 구현 되었다. 실험을 통하여 확인한 결과, 제안된 메커니즘에 의하여 빅리틀 아키텍처의 Linux CFS는 태스크를 코어에 할당할 때 코어의 사용률을 예측함으로써, 에너지 소비를 최적화 시킴을 알았다. CPU-intensive한 workload를 사용하여 실험 시, 기존 대비 최대 11.35%의 에너지 소비가 감소하였다. 또한 Android 응용 프로그램인 소프트웨어 디코더 실험 시, 기존 대비 동일한 QoS를 유지하면서 7.35% 에너지 소비가 감소하였다.

전체 코어 사용 방식을 위한 공정할당 스케줄링 기법은 ARM사의 Versatile Express TC2 board에 구현하였다. 한 개의 태스크로 이루어진 벤치마크를 여러 개 복사해서 시스템에 부가한 결과와, 여러 개의 태스크로 이루어진 벤치마크를 실험한 결과 모두에서, 클러스터 내부의 태스크간 최대 SVR 차이가 상수 값 이내로 제한됨을 확인하였다. 또한 동일한 CPU-intensive 태스크를 여러 개 시스템에 부가한 결과, 완료시간

편차가 기존 대비 56% 줄어 들었다. 이는 태스크들의 상대적인 진척도가 기존 대비 훨씬 더 비슷하게 유지됨을 직관적으로 보여주고 있다.

본 연구를 토대로, 다음과 같은 방법으로 비대칭 멀티코어 아키텍처에 최적화된 스케줄링 기법 연구를 확장할 수 있다. 첫째, 빅리틀 아키텍처가 제공하는 코어 타입 선택 방식에 최적화된 스케줄링 기법을 구현하고자, 본 연구에서는 태스크의 사용률 추정기를 사용하였다. 실험을 통하여 확인했을 때, 두 가지 제안된 추정기 중 태스크가 코어에 할당될 경우 그 사용률을 예측하는 추정기가 에너지 소비 감소 효과가 더 큼을 알았다. 따라서 이 추정기를 더욱 개선할 경우, 더 큰 에너지 소비 감소 효과를 얻을 수 있을 것으로 예측된다.

둘째, 제안된 공정할당 스케줄링 기법은 성능저하를 최소화 하고자, 기존의 Linaro 스케줄링 프레임웍의 클러스터간 태스크 이주정책을 그대로 사용하였다. 그리고 클러스터 내부의 태스크간 최대 SVR 값 차이를 상수 값 이내로 제한함으로써, 시스템 전체적으로는 최대 SVR 값 차이를 매우 작은 값으로 유지하였다. 향 후, 시스템 전체적으로 최대 SVR 값 차이를 작은 상수 값 이내로 제한시키는 연구가 필요하다. 이때, 클러스터간 태스크 이주정책에 대한 수정은 반드시 필요하다. 이에 대한 성능저하 최소화 기법이 연구된다면, 가장 완벽한 공정할당 스케줄링 기법의 결실을 얻을 것으로 기대된다.

참고 문헌

[1] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn, “Operating system support for overlapping-ISA heterogeneous multicore architectures,” in In Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA), Jan. 2010, pp.1–12.

[2] K. V. Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, “Fairness-aware scheduling on single-ISA heterogeneous multi-cores,” in Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT) Sep., 2013, pp. 177-188.

[3] S. Huh, J. Yoo, M. Kim, and S. Hong, “Providing fair share scheduling on multicore cloud servers via virtual runtime-based task migration algorithm,” in IEEE 32nd International Conference on Distributed Computing Systems (ICDCS), Jun. 2012, pp. 606-614.

[4] M. Kim, K. Kim, J. Geraci, and S. Hong, “Utilization-aware load balancing for the energy efficient operation of the big.LITTLE processor,” in In Proceedings of the Conference on Design, Automation & Test in Europe (DATE), Mar. 2014, pp. 1-4.

[5] ARM. (2015) High-performance applications processing for mobile and enterprise markets. [Online]. Available: http://www.arm.com/products/processors/cortex-a/index.php

[6] ARM. (2015) big.LITTLE technology, hardware requirements. [Online]. Available: http://www.arm.com/products/processors/technologies/biglittleprocessing.php

[7] T. Li, D. Baumberger, and S. Hahn, “Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin,” ACM SIGPLAN Notice, vol. 44, no. 4, pp. 65-74, Apr. 2009.

[8] A. K. Parekh and R. G. Gallagher, “A generalized processor sharing approach to flow control in integrated services networks: the multiple node case,” IEEE/ACM Transactions on Networking (TON), vol. 2, no. 2, pp. 137-150, Apr. 1994.

[9] ARM. (2012) Coretile express technical reference manual. [Online]. Available: http://www.arm.com/files/pdf/DDI0503B v2p ca15 a7 reference manual.pdf

[10] B. Caprita, W. C. Chan, J. Nieh, C. Stein, and H. Zheng, “Group ratio round-robin: O (1) proportional share scheduling for uniprocessor and multiprocessor systems,” in In Proceedings of the USENIX Annual Technical Conference, Apr. 2005, pp. 337-352.

[11] A. Chandra, M. Adler, P. Goyal, and P. Shenoy, “Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors,” in In Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation, vol. 4, Oct. 2000.

[12] A. Srinivasan and J. H. Anderson, “Fair scheduling of dynamic task systems on multiprocessors,” Journal of Systems and Software, vol. 77, no. 1, pp. 67-80, Jul. 2005.

[13] C. Kolivas. (2010) BFS scheduler. [Online]. Available: http://ck.kolivas.org/patches/bfs/bfs-faq.txt

[14] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, “Quincy: fair scheduling for distributed computing clusters,” in In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP), Oct. 2009, pp. 261-276.

[15] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, “Choosy: Maxmin fair sharing for datacenter jobs with constraints,” in In Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys), Apr. 2013, pp. 365-378.

[16] B. Caprita, J. Nieh, and C. Stein, “Grouped distributed queues: distributed queue, proportional share multiprocessor scheduling,” in In Proceedings of the 25th Annual ACM symposium on Principles of Distributed Computing (PODC), Jul. 2006, pp. 72-81.

[17] C. Shih, J. Wei, S. Hung, J. Chen, and N. Chang, “Fairness scheduler for virtual machines on heterogonous multi-core platforms,” ACM SIGAPP Applied Computing Review, vol. 13, no. 1, pp. 28-40, Mar. 2013.

[18] S. Hofmeyr, J. A. Colmenares, C. Iancu, and J. Kubiatowicz, “Juggle: proactive load balancing on multicore computers,” in In Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC), Jun. 2011, pp. 3-14.

[19] S. Hofmeyr, C. Iancu, and F. Blagojevic, “Load balancing on speed,” ACM SIGPLAN Notice, vol. 45, no. 5, pp. 147-158, May 2010.

[20] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, “Efficient operating system scheduling for performance-asymmetric multi-core architectures,” in In Proceedings of Supercomputing: the Conference on High Performance Computing Networking, Storage and Analysis (SC), Nov. 2007, pp. 1-11.

[21] V. Kazempour, A. Kamali, and A. Fedorova, “AASH: an asymmetry aware scheduler for hypervisors,” in In Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Mar. 2010, pp. 85-96.

[22] Y. Kwon, C. Kim, S. Maeng, and J. Huh, “Virtualizing performance asymmetric multi-core systems,” in In Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA), Jun. 2011, pp. 45-56.

[23] S. Huh, J. Yoo, and S. Hong, “Improving interactivity via VT-CFS and framework-assisted task characterization for Linux/Android smartphones,” in Embedded and Real-Time Computing Systems and Applications (RTCSA), Aug. 2012, pp. 250-259.

[24] S. Huh, J. Yoo, and S. Hong, “Cross-layer resource control and scheduling for improving interactivity in Android,” Software: Practice and Experience, 2014.

[25] PARSEC benchmark suites. [Online]. Available: http://parsec.cs.princeton.edu/index.htm

[26] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: characterization and architectural implications,” in In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT), Oct. 2008, pp. 72-81.

[27] “Advances in big.little technology for power and energy savings,” [Online]. Available: http://www.thinkbiglittle.com/

[28] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, “Singleisa heterogeneous multi-core architectures: The potential for processor power reduction,” in Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, 2003.

[29] M. Hilzinger. Con kolivas introduces new bfs scheduler. http://www.linux-magazine.com/Online/News/Con-Kolivas-Introduces - New-BFS-Scheduler/. (2009)

[30] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, “Hass: a scheduler for heterogeneous multicore systems,” SIGOPS Oper. Syst. Rev., vol. 43, no. 2, pp. 66-75, Apr. 2009.

[31] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, “Scheduling heterogeneous multi-cores through performance impact estimation (pie),” in Computer Architecture, 2012 39th Annual International Symposium on, pp. 213-224.

[32] L. Sawalha, S. Wolff, M. Tull, and R. Barnes, “Phase-guided scheduling on singleisa heterogeneous multicore processors,” in

Digital System Design, 2011 14th Euromicro Conference on, 2011, pp. 736-745.

[33] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov, “A comprehensive scheduler for asymmetric multicore systems,” in EuroSys, 2010, pp. 139-152.

[34] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, “Pack & cap: adaptive dvfs and thread packing under power caps,” in Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, 2011, pp. 175-185.

[35] X. Li, G. Yan, Y. Han, and X. Li, “Smartcap: User experience-oriented power adaptation for smartphone`s application processor,” in Design, Automation Test in Europe Conference Exhibition, 2013, pp. 57-60.

[36] “CPU frequency and voltage scaling code in the linux(tm) kernel,” [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

[37] “Corelink cci-400 cache coherent interconnect,” [Online]. Available: http://www.arm.com/products/system-ip/interconnect/corelink-cci-400.php

[38] M. Poirier, “In kernel switcher,” [Online]. Available: http://events.linuxfoundation.org/images/stories/slides/elc2013 poirier.pdf

[39] “This is the cfs scheduler,” May 2007. [Online]. Available: http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt

[40] Thannirmalai Somu Muthukaruppan, Mihai Pricopi, Vanchinathan Venkataramani, Tulika Mitra, Sanjay Vishin, “Hierarchical power management for asymmetric multi-core in dark silicon era,” in Proceedings of the Design Automation Conference, 2013.

[41] Linaro, [Online]. Available: https://www.linaro.org/

[42] Paul Turner. 2013. Per-entity load tracking. [Online]. Available: https://lwn.net/Articles/531853/. 2013

[43] Paramveer S. Dhillon, Dean P. Foster, Sham M. Kakade, Lyle H. Ungar, “A Risk Comparison of Ordinary Least Squares vs Ridge Regression,” Journal of Machine Learning Research, 2013

[44] Myungsun Kim, Soonhyun Noh, Sungju Huh, and S. Hong. 2015. “Fair-share Scheduling for Performance-asymmetric Multicore Architecture via Scaled Virtual Runtime,” In Proceedings of the 21st international conference Embedded and Real-Time Computing Systems and Applications (RTCSA). 2015.

[45] SPEC CPU2006 benchmark suites. [Online]. Available: https://www.spec.org/cpu2006

[46] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. SIGARCH Comput. Archit. News 34, 4 (September 2006), 1-17. DOI=http://dx.doi.org/10.1145/1186736.1186737

Abstract

Fair-share and Energy-efficient Scheduling in Performance-asymmetric Multicore Architecture

Myungsun Kim
School of Electrical Engineering and Computer Science
The Graduate School
Seoul National University

As users begin to demand applications with superior user experience and high service quality, asymmetric multicore processors are increasingly adopted in embedded systems due to their architectural benefits in improved performance and power savings. Those asymmetric multicore processors which are particularly designed for embedded systems often take the form of performance-asymmetric and single-ISA(Instruction Set Architecture) multicore architecture. Such architecture consists of high-performance cores and energy-efficient cores. By assigning compute-intensive workloads to high-performance cores while others to energy-efficient cores, we can achieve a performance goal within a limited power and area budget.

There are two distinct modes in operation for this architecture: i) *Switching* and ii)

Global. In the *Switching* mode, each high-performance core is paired with an energy-efficient core and only one core per high-performance/energy-efficient pair can be running at any given time. In the *Global* mode, all cores are available for execution at any time and thus the architecture can exhibit the most flexibility in scheduling. ARM's big.LITTLE architecture is a representative example of such performance-asymmetric multicore architecture which enables a varying number of big and little cores to be utilized at the same time. Linux kernel is widely used for this architecture and its CFS (completely fair scheduler) supports both *Switching* mode and *Global* mode.

Unfortunately, the present CFS for each operating mode has several limitations. First, the CFS with *Switching* mode support neither distinguishes between the core types nor considers how heavily a core is being used when assigning tasks. Therefore, it runs the chance of running a task unnecessarily on a high-frequency core, unnecessarily increasing the core frequency or causing unneeded little cores to big cores transitions. These all cause the processor to consume more energy than necessary.

Second, the present CFS with *Global* mode support falls short of expectations when it comes to fair-share scheduling. It simply lets runnable tasks share physical CPU time in proportion to their weights. In reality, in an asymmetric multicore system, computing power significantly varies between a big core and a little core. Thus, the physical CPU time given to a task should be scaled according to the computing power of the hosting core. In addition, the CFS tries to achieve fair-share scheduling by minimizing the virtual runtime differences among all runnable tasks. It achieves this goal in per-core scheduling, whereas in multicore scheduling, it cannot guarantee fair-share scheduling as CFS's weight-based load balancing has

nothing to do with minimizing the virtual runtime differences.

To overcome abovementioned limitations, this thesis proposed two optimization methods for scheduling in performance-asymmetric multicore architecture. Especially, we took ARM's big.LITTLE architecture as a target system and presented solution approaches. First, we proposed a utilization-aware load balancing mechanism to provide the *Switching* mode with energy-efficient operation. To do so, we carefully analyzed the power management scheme of the big.LITTLE processor's port of Linux kernel and derived its state diagram representations. Based on our formulation, we incorporated the processor utilization factor into the Linux kernel's load balancing algorithm. As a result, our mechanism improved the Linux kernel's ability to assign tasks to cores in an energy-efficient manner without having to make it directly aware of the available core types.

Second, we proposed a solution approach for the fair-share scheduling in the *Global* mode. It uses scaled CPU time which reflects the relative performance of cores with respect to operating frequencies as well as different core types. Since it is a modified version of CFS, each task gets scaled CPU time in proportional to its weight. Thus, the weighted-based virtual runtime of original CFS is extended to the weight-based SVR (scaled virtual runtime) by adopting the scaled CPU time. We also presented SVR-based load balancing algorithm. It periodically makes tasks with larger SVRs run more slowly since they are allocated to the core with heavier load in the following balancing period. As a result, It bounds the SVR difference between any pair of tasks in the same cluster by a constant.

To evaluate the effectiveness of the proposed approaches, we have implemented both solutions on top of commercial products. We have implemented the utilization-aware load balancing mechanism into Galaxy S4. Experimental results

show that it can reduce energy consumption by 11.35% compared to the original Linux CFS. Our method's energy efficiency and computation performance depend on the utilization estimator, so better estimators could produce greater gains. Complexity of any estimator, however, must be simple enough so as to not burden the scheduler with its compute time, This trade-off between estimator complexity of implementation and the benefits provided by the increased complexity make further research into utilization estimators a promising area of future work. We have also implemented the solution approach for fair-share scheduling into ARM's Versatile TC2 board. Experimental results show that the maximum SVR difference in our solution was bounded by a constant while it diverges indefinitely in the original CFS. The standard deviation of finish time of multiple identical tasks under our approach is smaller by 56% than the original CFS. These results clearly demonstrate that our proposed approach can provide enhanced fairness in performance-asymmetric multicore architecture.

Keywords : Performance-asymmetric multicore, tsk scheduling, load balancing, low power scheduling, big.LITTLE architecture

Student Number : 2011-30218

공학박사 학위논문

비대칭 멀티코어 아키텍처용 공정성 보장 및 에너지 효율적인 스케줄링

Fair-share and Energy-efficient Scheduling in Performance-asymmetric Multicore Architecture

2016년 2 월

서울대학교 대학원

전기 · 컴퓨터공학부

김 명 선

비대칭 멀티코어 아키텍처용 공정성 보장 및 에너지 효율적인 스케줄링

Fair-share and Energy-efficient Scheduling in Performance-asymmetric Multicore Architecture

지도 교수 홍 성 수

이 논문을 공학박사 학위논문으로 제출함

2016 년 2 월

서울대학교 대학원
전기·컴퓨터 공학부
김 명 선

김 명 선의 공학박사 학위논문을 인준함
2016 년 2 월

위 원 장 : 김 태 환 (인)
부위원장 : 홍 성 수 (인)
위 원 : 심 규 석 (인)
위 원 : 엄 현 상 (인)
위 원 : 전 광 일 (인)

초 록

최근 우수한 사용자 체감과 질적으로 향상된 수준의 서비스를 위하여 스마트폰, 태블릿과 같은 임베디드 시스템에서 비대칭 멀티코어 아키텍처의 사용이 크게 증가되고 있다. 이는, 이러한 아키텍처가 임베디드 시스템의 제한된 면적과 소비전력 환경하에서 주는 하드웨어적 이점 때문이다. 임베디드 시스템에 사용되는 비대칭 멀티코어 아키텍처는 서로 다른 특성을 가지는 두 가지 타입의 코어들로 이루어진다. 첫 번째 타입의 코어는 높은 성능과 낮은 에너지 효율성을 특징으로 하고, 두 번째 타입의 코어는 낮은 성능 대신 높은 에너지 효율성을 특징으로 한다.

비대칭 멀티코어 아키텍처를 운용하는 방법에는, 각 코어들을 사용하는 방법에 따라서 두 가지 종류가 있다. (1) 한 개의 높은 성능 코어와 한 개의 에너지 효율적인 코어를 하나의 pair로 형성한 후, 그 중 한 개의 코어만 동작하게 하는 코어 타입 선택 방식과 (2) 시스템의 모든 코어를 동시에 사용할 수 있는 전체 코어 사용 방식으로 운용된다. Linux kernel은 이러한 비대칭 멀티코어 아키텍처에 가장 널리 쓰이는 운영체제이며, CFS(completely fair scheduler)를 사용하여 태스크들을 스케줄링 한다. 또한, CFS는 비대칭 멀티코어 아키텍처의 코어 타입 선택 방식과 전체 코어 사용 방식에 맞는 스케줄링 프레임웍을 제공하고 있다.

하지만 현재 제공되고 있는 두 가지 코어 사용 방식을 위한 스케줄링 프레임웍에는 다음과 같은 문제점이 있다. 첫째, 코어 타입 선택 방식에서의 스케줄링 프레임웍은 비대칭 멀티코어간 부하분산 시

태스크의 가중치(weight)만 고려하여 부하분산을 수행한다. 이로 인하여 필요이상으로 코어의 동작 주파수를 상승시키거나, 에너지 효율적인 코어에서 수행해도 충분한 태스크가 고성능 코어에서 운용되어 소비전력을 크게 증가시키는 문제점을 야기시킨다.

둘째, Linux kernel의 CFS는 virtual runtime을 통하여, 태스크들에게 가중치에 비례하는 CPU 사용 시간을 부여한다. 이때, 태스크의 virtual runtime 산정 시, 현재 태스크를 수행하는 코어의 상태(코어 타입 혹은 동작 주파수 등)를 고려하지 않는다. 이로 인하여 공정한 CPU 사용 시간을 태스크들에게 부여하지 못한다. 또한, CFS는 태스크들의 virtual runtime을 동일하게 만들려고 노력한다. 이는 태스크들의 상대적인 수행 정도를 비슷하게 유지시키기 위해서이다. 하지만 이는 개별 코어에서는 유지되나, 시스템 전체적으로 비슷하게 유지 되지 않는다. 이는 시간이 지남에 따라서 태스크간 virtual runtime 차이를 증가시켜, 태스크간 상대적인 수행 정도 차이가 더욱 더 커지게 하는 문제점을 발생시킨다.

본 학위논문은 비대칭 멀티코어 아키텍처가 지원하는 코어 타입 선택 방식과 전체 코어 사용 방식에 최적화된 스케줄링 기법을 제안한다. 첫째, 본 연구는 코어 타입 선택 방식의 저전력 운용 목적에 맞는 스케줄링 기법을 제안한다. 이를 위하여 먼저, Linux kernel이 비대칭 멀티코어 아키텍처를 위하여 제공하는 DVFS(Dynamic Voltage and Frequency Scaling) 정책을 정확히 분석한다. 분석된 결과를 토대로 정확한 동작을 모델링하고, 이를 제안하는 스케줄링 기법에 반영한다. 이 기법은 부하분산 시 태스크의 가중치뿐만 아니라, 코어의 사용률을 고려하여 부하를 분산시킨다. 이를 통하여 성능 저하를 최소화 하면서

주파수 상승을 억제하고, 동시에 고성능 코어를 최대한 적게 사용하는 저전력, 에너지 효율적인 스케줄링을 수행한다.

둘째, 본 연구는 전체 코어 사용 방식에 적합한 공정할당 스케줄링 방식을 제안한다. 이는 코어의 상태를 반영한 스케일된 CPU 시간을 구한다. 이를 CFS의 virtual runtime에 반영하고, SVR (scaled virtual runtime)로 확장한다. 또한 각각의 고성능 코어로 이루어진 클러스터와 에너지 효율적인 코어로 이루어진 클러스터 내부에서, 모든 태스크들의 SVR 차이를 일정한 상수 크기로 제한시킨다. 이를 통하여 클러스터 내부의 모든 태스크들의 상대적 진척 정도를 비슷하게 유지시킨다. 결과적으로, 시스템 전체적인 공정할당 스케줄링을 수행하도록 한다.

본 연구에서 제안하는 두 가지 스케줄링 기법들의 효용성을 입증하기 위해서, 실제 상용으로 출시된 비대칭 멀티코어 아키텍처 기반 제품에 이들을 구현하였다. ARM사의 빅리틀 아키텍처를 대상 시스템으로 하였으며, 이는 임베디드 시스템에서 가장 대표적으로 사용되는 비대칭 멀티코어 아키텍처이다. 저전력 스케줄링 기법은 코어 타입 선택 방식이 지원되는 Galaxy S4 Android 스마트폰에 구현되었다. CPU-intensive한 부하를 사용하여 실험한 결과, 기존 대비 최대 11.35%의 에너지 소비가 감소하였다. 또한 Android 응용프로그램을 이용하여 실험 시, 기존 대비 동일한 QoS를 유지하면서 7.35% 에너지 소비가 감소하였다. 이러한 실험 결과는 비대칭 멀티코어에서 제안된 스케줄링 기법이 에너지 효율적임을 증명한다.

공정할당 스케줄링 기법은, 전체 코어 사용 방식이 지원되는 ARM사의 Versatile Express TC2 board에 구현하였다. 실험 결과,

클러스터 내부의 태스크간 SVR 차이가 상수 값 이내로 제한됨을 확인하였다. 또한 SVR 차이를 작게 만드는 것이 공정할당 스케줄링에 어떠한 영향을 실질적으로 미치는지 알기 위해서, 동일한 태스크를 여러 개 생성한 후 그들의 완료시간 표준 편차를 측정하였다. 실험 결과, 기존 대비 완료시간 표준편차가 56% 줄어 들었다. 이러한 실험 결과는 본 논문에서 제안된 스케줄링 기법이 태스크들에게 더 공정한 CPU 시간을 부여함을 직관적으로 알 수 있게 한다.

주요어: 비대칭 멀티코어, 태스크 스케줄링, 멀티코어 부하분산, 저전력 스케줄링, 빅리틀 아키텍처

학 번: 2011-30218

목 차

그림 목차

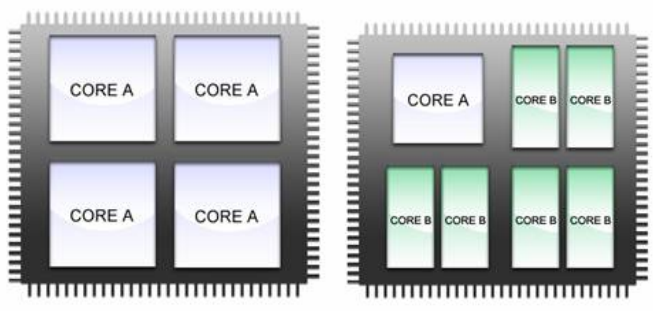
7

표 목차

제 1 장 서 론

스마트폰, 태블릿과 같은 현대의 임베디드 시스템은 복잡해짐과 동시에 LTE와 같은 고속 모뎀 통신, 다양한 센서 모듈을 통한 빠른 주변환경 인식, 고해상도 디스플레이, 대용량 멀티미디어 컨텐츠 실행 기능 등을 갖추고 있다. 이러한 경향으로 인하여, 임베디드 시스템 사용자들은 훨씬 더 향상된 사용자 체감과 높은 질의 서비스를 요구하고 있다. 이를 위하여 임베디드 시스템은 고성능 멀티코어를 사용하기 시작했다. 또한, 향상된 반도체 기술에 힘입어 갈수록 코어의 숫자는 늘어가고 코어의 동작 주파수는 높아지고 있다.

대부분의 고성능 멀티코어 아키텍처는 그림 1의 (a)와 같은 대칭 멀티코어(symmetric multicore processor)로 구성되고, 각 코어에는 수퍼스칼라(superscalar), 다 계층 파이프라인 스테이지(multi-level pipeline



(a) (b)

그림 1. 멀티코어 아키텍처: (a) 대칭 멀티코어, (b) 비대칭 멀티코어

stage), 비순차적 수행(out of order)과 같은 기술들이 내제되어 있다. 하지만 이러한 기술들은 임베디드 시스템에서 사용되는 SoC(system on a chip)에서 상대적으로 넓은 면적을 차지하며, 높은 소비전력을 나타낸다. 코어 수가 증가되면서, 이러한 특성은 SoC의 면적과 소비전력 측면에서 문제로 부각되었다 [1][5][6].

이러한 문제점을 해결하기 위해서 임베디드 시스템 개발자들은 그림 1의 (b)에 나타낸 비대칭 멀티코어(performance-asymmetric multicore processor) 아키텍처를 채택하기 시작하였다. 비대칭 멀티코어 아키텍처는 같은 명령어 집합(single-ISA: single instruction set architecture)을 사용하고 두 가지 종류의 코어로 이루어져 있다: 1) 차지하는 면적이 크고 높은 주파수로 운용되는 고성능 코어, 2) 면적이 작고 주로 낮은 주파수로 운용되는 에너지 효율적인 코어. 고성능을 필요로 하는 응용프로그램과 백그라운드 혹은 IO 집중적인 응용프로그램들을 각각 고성능 코어와 에너지 효율적인 코어에 할당함으로써, 같은 코어 수의 대칭 멀티코어 아키텍처보다 향상된 단위 전력당 성능(performance/watt)을 나타낼 수 있다.

비대칭 멀티코어 아키텍처의 동작 모드는 1) 한 개의 고성능 코어와 한 개의 에너지 효율적 코어를 pair로 구성하여 둘 중 한 개만 동작시켜서 저전력 효과를 극대화하는 코어 타입 선택 방식과 2) 모든 코어를 동시에 사용할 수 있고, 또한 모든 코어가 독립적으로 제어될 수 있는 전체 코어 사용 방식 두 가지가 있다. 이러한 두 가지 동작 모드 각각의 장점을 최대한 활용하기 위해서는 운용되는 운영체제의 스케줄링 방식이 각각의 모드에 맞게 최적화 되어야 한다. 코어 타입 선택 방식에서의 스케줄러는 코어에 태스크를 할당할 때, 고성능 코어와 에너지 효율적인 코어 중, 해당 시점에 어느 코어에 할당해야 가장 저전력 효과가 있는지를 알

아야 한다. 전체 코어 사용 방식에서의 스케줄러는 멀티코어를 사용함에 있어서, 각 태스크에게 공정성을 보장해야 한다. 즉, 스케줄러는 태스크를 코어에 할당할 때, 공정하게 고성능 코어와 에너지 효율적인 코어를 태스크들이 사용할 수 있도록 설계되어야 한다.

본 논문에서는 이러한 비대칭 멀티코어 아키텍처의 두 가지 동작모드에 최적화된 스케줄링 알고리즘을 제안한다. 또한, 제안된 기법들의 성능을 해당 동작 모드에 적용해 그 실효성과 효과를 검증한다. 본 장의 1절에서는 비대칭 멀티코어 아키텍처의 두 가지 동작모드에 대하여 기존의 스케줄링 방식이 가지고 있는 문제점 제시 및 본 연구에 대한 동기를 기술한다. 이어서, 2절에서는 본 연구에서 제시한 알고리즘들의 기여에 대하여 논한다. 마지막으로, 3절은 본 논문 전체의 구성을 설명한다.

제 1 절 연구 동기

고성능 멀티미디어 컨텐츠 프로세싱 기능 및 빠른 사용자 응답 특성 등을 임베디드 시스템이 가지는 제한된 소비전력 환경하에 수행하려면, 비대칭 멀티코어 아키텍처가 가지는 하드웨어적 특성을 최대한 인지한 스케줄링 기법이 필요하다. 구체적으로, 본 연구에서는 비대칭 멀티코어 아키텍처의 두 가지 동작모드에서, 현재 사용되는 스케줄링 방식의 문제점을 설명하고, 각각의 문제점에 대한 해결책을 제시한다. 본 연구논문의 목적은 비대칭 멀티코어 아키텍처의 두 가지 동작 모드에 최적화된 스케줄링 알고리즘들을 제안하고, 각각의 동작 모드에서 제안된 알고리즘의 실효성을 증명하는 것이다. 비대칭 멀티코어 아키텍처의 두 가지 동작모드에 대한 스케줄링 알고리즘을 연구하게 된 동기를

설명하면 다음과 같다.

1.1 코어 타입 선택 방식에서의 저전력 스케줄링 최적화

비대칭 멀티코어 아키텍처가 코어 타입 선택 방식에서 동작할 때, 각 pair에 존재하는 고성능 코어와 에너지 효율적인 코어 중 한 개의 코어만 사용된다. 이때, 사용되는 코어의 결정은 현재 사용되는 코어의 사용 정도에 따라 결정된다. 코어의 사용률이 어느 한계 보다 높을 때는 현재 주파수보다 더 높은 주파수로 변경하거나, 혹은 에너지 효율적인 코어에서 고성능 코어로 스위칭된다. 반대로 어느 한계보다 낮을 때는 현재 주파수보다 낮은 주파수로 변경하거나, 혹은 고성능 코어에서 에너지 효율적인 코어로 스위칭된다. 따라서 스케줄러가 태스크를 어떤 코어에 할당하는지에 따라 코어가 스위칭되거나 혹은 동작 주파수의 변경이 생긴다. 태스크를 할당해도 사용률이 어느 한계를 넘지 않아, 현재의 동작 주파수를 유지할 수 있는 코어를 선택하거나 에너지 효율적인 코어에서 고성능 코어로 필요이상으로 스위칭하는 현상을 막는다면 소비전력 측면에서 최적화를 달성할 수 있다.

불행히도 비대칭 멀티코어 아키텍처의 코어 타입 선택 방식에서, 현재의 Linux CFS는 태스크를 할당할 때, 코어의 타입을 구분하거나 현재 코어가 얼마나 사용되고 있는지 고려하지 않는다. 이로 인하여 불필요한 동작 주파수 상승을 야기할 수 있다. 또한, 에너지 효율적인 코어에서 고성능 코어로 스위칭 동작을 과도하게 발생시킬 수 있다. 결과적으로, 이는 시스템 전체 소비전력을 필요이상으로 증가시키는 문제점을 야기시킨다. 이러한 문제점을 해결하기 위하여, Linux CFS의 태스크 할당정책에 현재 동작중인 코어의 사용 정도를 정량화하여

반영해야 한다. 이는 Linux CFS가 코어 타입 선택 방식을 위하여 에너지 효율적인 스케줄링 방식으로 개선되었다는 뜻이 된다.

1.2 전체 코어 사용 방식에서 공정할당 스케줄링 최적화

전통적인 임베디드 시스템에서 사용되는 멀티코어 스케줄링은 빠른 사용자 반응성 혹은 향상된 단위시간당 데이터 처리량에 집중되었다. 최근에 공정할당(fair-share) 스케줄링은 실제 활용 사례와 여러 가지 많은 연구결과들에서 중요한 기술로 다루어졌다. 공정할당 스케줄링은 태스크들에게 부여된 가중치(weight)에 비례하여 태스크들이 코어를 사용하게 하는 방식이다. 이러한 스케줄링 정책은 운영체제 수준에서 QoS(Quality-of-Service)를 제공한다 [2][3]. 현재의 공정할당 스케줄링은 대부분이 대칭 멀티코어 아키텍처를 위하여 개발되었고, 비대칭 멀티코어 아키텍처용 공정할당 스케줄링 알고리즘은 아직 개발이 미흡하여, 늘어나는 추세의 비대칭 멀티코어 아키텍처에 적절히 대응하지 못하고 있다.

앞서 설명한 비대칭 멀티코어가 주는 아키텍처적 이로운 점에도 불구하고, 효과적인 공정할당 스케줄링 정책을 개발하는 것은 이러한 아키텍처가 본연적으로 가지고 있는 코어간 성능 비대칭성 때문에 쉽지 않은 일이다. 대칭 멀티코어 아키텍처 환경에서는, 각 코어의 처리 능력이 동일하다. 그래서 공정할당 스케줄러는 실행 가능한 태스크들에게 CPU 시간을 할당할 때, 단순히 각 태스크들의 가중치에 비례하게 부여하면 된다. 이와 대비하여 비대칭 멀티코어 아키텍처에서는 고성능 코어와 에너지 효율적인 코어간의 처리 능력이 많이 다르다. 이러한 이유로 한 태스크에게 주어지는 CPU 시간은 태스크가 사용하는 코어의

처리 능력에 따라서 스케일(scale) 되어야 한다.

또한, CFS는 태스크들의 virtual runtime을 동일하게 유지하려고 노력한다. 하지만 이는 개별 코어에 국한되고, 시스템 전체적으로 태스크간 virtual runtime은 비슷하게 유지 되지 않는다. 이는 멀티코어간 부하분산을 virtual runtime을 기반으로 하지 않고, 태스크들의 가중치에 기반하여 부하를 분산하기 때문이다.

비대칭 멀티코어 아키텍처의 전체 코어 사용 방식에서, Linux CFS는 태스크들의 CPU 시간을 계산할 때, 코어의 처리 능력을 고려하지 않고 대칭 멀티코어 아키텍처와 동일한 방식으로 계산한다. 이에, 코어의 처리 능력을 고려한 스케일된 CPU 시간을 Linux CFS에 반영하고, 시스템 전체적으로 태스크간 virtual runtime을 비슷하게 유지한다면, Linux CFS가 비대칭 멀티코어 아키텍처를 위한 공정할당 스케줄링을 수행한다는 뜻이 된다.

제 2 절 논문의 기여

본 학위논문에서는 비대칭 멀티코어 아키텍처의 코어 타입 선택 방식과 전체 코어 사용 방식에 사용되는 최적화된 스케줄링 기법 연구에 대한 내용을 기술한다. 구체적으로, 코어 타입 선택 방식에서는 저전력 스케줄링 최적화를, 전체 코어 사용 방식에서는 코어간의 성능 비대칭성을 스케줄러에 반영하고, 최적화된 공정할당 스케줄링을 수행한다.

본 학위논문의 학술적, 기술적 기여를 요약하면 다음과 같이 나타낼 수 있다.

- **비대칭 멀티코어 아키텍처용 CFS 분석**

본 연구에서는 비대칭 멀티코어 아키텍처를 지원하기 위한 Linux CFS를 분석하고, 코어 타입 선택 방식과 전체 코어 사용 방식에서 각각의 모드를 위해서 어떤 기능들을 추가했는지 정리하였다. 분석 결과, 코어 타입 선택 방식에서는 코어내부에서의 스케줄링과 코어간 부하분산 정책은 기존의 Linux CFS를 유지하고, CPU 드라이버 소프트웨어를 통하여 고성능 코어와 에너지 효율적인 코어간 스위칭을 수행함을 알았다.

전체 코어 사용 방식에서는 코어 내부의 스케줄링은 기존 CFS를 유지하되, 코어간 부하분산 정책은 기존의 CFS의 부하분산정책을 같은 코어 타입간의 부하분산 정책에 적용하였다. 서로 다른 코어 집합간의 부하분산 정책은 별도의 태스크 이주 정책을 사용하고 있음을 알았다. 또한, 이러한 부하분산 정책들로 인하여, 공정할당 측면에서의 최적화되지 못한 점을 발견하였다.

- **코어 타입 선택 방식으로 동작 시 저전력 스케줄링 측면에서의 문제점 파악**

본 연구에서, 현재의 Linux CFS가 코어 타입 선택 방식에서 스케줄링을 수행할 때, 코어의 타입이나 주파수를 고려하지 않고 태스크들의 가중치만 고려하여 부하분산을 수행함을 찾았다. 이러한 부하분산 정책은 동작 주파수를 불필요하게 상승 시킬 수 있다. 또한, 에너지 효율적인 코어에서 고성능 코어로 스위칭 횟수를 증가시킬 수 있다. 이러한 점들은 전체적인 에너지 소비를 증가시킨다.

- **코어 타입 선택 방식에서의 DVFS 모델**

코어 타입 선택 방식에서 저전력을 고려한 부하분산 정책의 목표를 달성하기 위하여 본 연구에서는 먼저, 코어의 주파수 변동과 고성능 코어와 에너지 효율적인 코어간 스위칭을 발생시키는 메커니즘을 분석하였다. 이

를 위하여 Linux ONDEMAND governor의 DVFS 모델을 상태도(State Diagram)에 나타내었다 [4]. 코어 사용률과 주파수 크기를 각각 3단계 및 2단계로 정의하고, 이를 (주파수, 사용률) 형태로 나타내어 코어의 상태를 표현하였다. 코어의 상태변화를 나타냄으로써 단순화된 DVFS 모델을 얻을 수 있었다.

- **코어 타입 선택 방식에서의 사용률인지 기반 부하분산 (utilization-aware load balancing) 알고리즘**

상태도의 DVFS 모델을 통해서, 코어의 동작 주파수 변경과 코어 타입 간 스위칭의 정확한 메커니즘을 분석할 수 있었다. 분석된 결과를 사용하여, 본 연구에서는 코어 타입 선택 방식에 적합한 사용률인지 기반 부하분산 알고리즘(utilization-aware load balancing)을 제안하였다. 이 알고리즘은 각 코어의 실행 큐(run-queue)에 있는 태스크들을 분산시킨다. 어떤 태스크가 시스템의 준비 큐(wait-queue)에서 빠져 나와 할당될 실행 큐를 찾을 때, 가장 사용률이 낮은 코어에 할당하게 함으로써 코어의 주파수 상승이나 에너지 효율적인 코어에서 고성능 코어로의 스위칭 발생 빈도를 최대한 줄인다. 실험 결과를 토대로, 기존 Linux CFS 대비 최대 11.35% 소비 전력 감소를 나타냄을 보인다.

- **코어 타입 선택 방식에서의 사용률 기반 추정기(utilization-based estimator)**

사용률인지 기반 부하분산 알고리즘이 준비 큐에서 빠져 나온 태스크를 실행 큐에 할당할 때, 이 태스크로 인한 해당 코어의 사용률 증가를 예측할 필요가 있다. 이를 위하여 본 연구에서는 코어의 사용률기반 추정기(utilization-based estimator)를 제안한다. 이 추정기를 통하여 사용률 증가가 가장 작은 코어를 알아내고, 태스크를 그 코어의 실행 큐에 할당한다.

■ **전체 코어 사용 방식으로 동작 시 공정할당 스케줄링 측면에서의 문제점 파악**

첫째, 전체 코어 사용 방식으로 동작 시 Linux CFS는 코어간 성능 비대칭성을 고려하지 않는 CPU 시간을 사용하며, 이를 CFS의 virtual runtime 산정 시 그대로 사용함을 알았다. 이는 태스크간의 CPU 공정할당(fair-share)성을 훼손한다는 것을 밝혔다.

둘째, CFS는 태스크들의 virtual runtime을 동일하게 유지하려고 노력한다. 이는 태스크들의 상대적인 진척도를 비슷하게 유지시키기 위함이다. 하지만 이러한 노력은 개별적인 코어에서는 유지되나, 시스템 전체적으로 유지 되지 않는다. 이러한 이유로, 시스템의 수행 시간이 늘어남에 따라서 태스크간 virtual runtime 차이는 증가된다. 이는 태스크간 상대적인 수행 정도 차이가 더욱 더 커지게 하는 문제점을 발생시킨다.

■ **전체 코어 사용 방식을 위한 SVR(Scaled Virtual Runtime) 정의**

Linux CFS는 virtual runtime을 사용하여 태스크의 상대적 진척 정도를 나타낸다 [39]. 이는 태스크의 가중치에 반비례하고, CPU 시간에 비례한다. 이 때, CPU 시간은 대칭 멀티코어 아키텍처임을 가정하고 산정한다. 본 연구에서는 스케일된 CPU 시간을 제안한다. 이는 코어간 성능 비대칭성을 고려한 CPU 시간이다. 또한 같은 코어 타입에서도 주파수에 따라서 나타내는 성능이 다르다. 따라서 주파수에 따른 성능 차이도 스케일된 CPU 시간에 반영하였다. 이렇게 구해진 스케일된 CPU 시간을 Linux CFS의 virtual runtime 계산에 적용하였다. 이를 스케일된 virtual runtime 즉, SVR(scaled virtual runtime)로 정의하였다.

■ **전체 코어 사용 방식을 위한 SVR 기반 공정할당 스케줄링 (SVR-based fair-share scheduling)**

비대칭 멀티코어의 공정할당 문제를 해결하기 위해서, 본 학위논문에서는 SVR 기반 공정할당 스케줄링(SVR-based fair-share scheduling) 알고리즘을 제안한다. 본 연구는 태스크들의 상대적인 진척 정도를 SVR(scaled virtual runtime)로 정의하고, 이를 부하분산을 통해서 균등하게 맞춘다. 비대칭 멀티코어 아키텍처는 고성능 코어집합인 고성능 클러스터와 에너지 효율적인 코어집합인 클러스터로 이루어지며, 본 연구에서는 각각의 클러스터 내부에서 태스크들의 진척 정도 차이를 작은 상수로 제한시킨다. 이로써 공정할당 스케줄링의 최적화 목표를 달성한다.

수학적 분석을 통하여 임의의 두 개의 태스크간 SVR 차이 값은 작은 상수 이내로 제한된다는 것을 밝혔다. 공정할당 스케줄링의 보다 직관적인 효과를 보기 위하여 여러 개의 같은 태스크들을 실행시키고 그들의 완료시간을 측정하였다. 그 결과, 기존 방식 대비, 태스크들의 완료시간 표준 편차가 56 %감소됨을 보였다.

제 3 절 논문 구성

본 학위논문의 나머지 부분의 구성은 다음과 같다. 2장에서는 관련 연구 및 배경 지식을 설명한다. 3장에서는 비대칭 멀티코어 아키텍처의 코어 타입 선택 방식에 최적화된 저전력 스케줄링 기법을 4장에서는 비대칭 멀티코어 아키텍처용 전체 코어 사용 방식에 최적화된 공정할당 스케줄링 기법을 설명한다. 5장은 본 연구에서 제안한 두 가지 스케줄링 알고리즘에 대한 실험적 검증을 기술하고, 아울러 수학적 분석을 통한 제안된 알고리즘의 실효성을 분석한다. 마지막 6장은 본 학위논문의 결론부이며, 향후 연구 방향에 대하여 기술한다.

제 2 장　　관련 연구 및 배경 지식

본 장에서는 본 학위논문의 토대가 되는 기존에 수행된 관련 연구들에 대해서 정리하여 기술한다. 또한, 본 연구를 수행함에 있어서 관련된 배경 지식을 소개한다. 구체적으로, 멀티코어 아키텍처의 스케줄링 관련 연구들 중 저전력 운용기법관련 기존 연구를 소개한다. 이어, 공정할당 스케줄링 관련 연구들을 기술한다. 최근 비대칭 멀티코어 아키텍처의 대표적인 예는 ARM사의 빅리틀 아키텍처이다. 따라서 본 연구에서는 이 아키텍처를 실험 대상으로 하였다. 본 장에서 다루는 배경 지식은 이의 하드웨어적, 소프트웨어적 특징들을 설명한다.

제 1 절　관련연구

1.1 멀티코어 아키텍처용 저전력 스케줄링

모든 코어들이 같은 명령어 집합(Single-ISA)을 사용하는 비대칭 멀티코어 아키텍처는 많은 연구들의 대상이 되어 왔다 [1][28]. 특히, 그들 중 대부분의 연구들은 이러한 아키텍처상의 프로세서들에게 최적화된 방법으로 태스크들을 할당하는 연구에 중심을 두었다 [30][31]. 또한 많은 연구들이 에너지 효율성 보다는 최대한의 컴퓨팅 성능을 낼 수 있는 스케줄링 기법에 대한 연구를 수행해 왔다 [32][33]. 하지만, 최근 들어, 에너지 효율성에 대한 관심이 연구 주제로 채택되기 시작했다.

Cochran은 최적화된 DVFS 운용과 태스크할당을 위한 제어 기법을 제안했다 [34]. 이 연구에서는 온도, 소비전력 데이터, 성능 측정용 카운터 등을 이용하여 제한된 전력 예산 하에 최대 성능을 내기 위한 방법을 제안하였다. Pack & Cap이라 불리는 이 방법은 태스크들을 패킹(Packing)하여 코어들에 할당하고, 일부 유휴(idle) 코어들은 Sleep상태에 진입하게 하여, 소비전력 상한선(Capping)내에서 성능 손실 없이 동작하게 한다.

SmartCap은 자가적응(self-adaptive)형 소비전력 제어 기법을 제안한다 [35]. 이 기법은 사용자 체험에 기반한 샘플링된 통계 데이터를 사용한다. 이를 사용하여 멀티코어 아키텍처가 가장 작게 사용할 수 있는 주파수를 응용 프로그램 별로 구한 후 이를 Linux의 DVFS [36]에 반영하였다. 그 결과 주파수를 필요 이상으로 상향 조정(over-provisioning)하는 현상을 막을 수 있다.

HPM(Hierarchical Power Management)은 빅리틀 아키텍처에서 소비전력과 온도를 성능과 관련시켜 제어하는 기법을 제안하고 있다 [40]. 이 기법은 태스크들의 QoS를 고려하여 코어의 주파수를 throttling시키는 방법을 사용한다. 이를 위하여, HPM은 PID(proportional integral derivative)기반 제어 기법을 사용하며, 허락된 소비전력과 칩 온도 하에서 QoS를 떨어뜨리지 않는 수준으로 코어의 주파수를 동작시킨다.

위에 열거된 [34][35][40]은 코어의 주파수를 직접 제어하여 소비전력 상승을 억제하는 방법을 사용하고 있다. 이와 달리 본 연구에서는 Linux kernel의 부하분산 기법을 개선하여 에너지 효율성을 높이는 방법을 사용한다. 본 연구에서의 주된 관심사는 현재 사용되는 Linux kernel의 부하분산 기법은 코어의 사용률을 고려하지 않는다는 점이다. 하지만 Linux kernel의 DVFS 모듈은 코어의 사용률에 따라서 동작 주파수나

코어 타입을 결정한다. 부하분산 기법과 DVFS 모듈간의 이러한 커뮤니케이션 부재는 비대칭 멀티코어 아키텍처의 최적화되지 못한 소비전력 결과를 초래한다.

1.2 멀티코어 아키텍처용 공정할당 스케줄링

공정할당을 위한 멀티코어 아키텍처용 스케줄링은 다양하고 폭 넓게 연구되어 왔다. 이 연구들의 대부분의 목적은 서버와 같은 시스템에서 사용자들에게 각각 구분된 성능을 보장하기 위함이다. 이들은 크게 중앙(centralized) 실행 큐 알고리즘 기반과 분산(distributed) 실행 큐 알고리즘 기반 스케줄링으로 분류될 수 있다. 중앙 실행 큐 알고리즘 방식은 시스템상에 한 개만 존재하는 실행 큐를 사용하기에 간단하고 직관적이다 [10][11][12][13][29].

분산 실행 큐 알고리즘 방식은 실행 큐를 차지하기 위한 태스크들의 경쟁을 피하기 위해서 코어마다 자신의 실행 큐를 갖게 한다. 특히 이러한 방식은 큰 규모의 시스템에서 더욱 효과적이다. 이 알고리즘은 각 코어 별로 다른 코어와 상관없이 스케줄링을 수행한다. 따라서 코어간 태스크 이동은 피할 수 없다. 만약 태스크 이동이 없다면, 각 코어들의 부하 크기 차이는 시간에 따라서 증가된다. 태스크 이동 정책에 따라서 분산 실행 큐 알고리즘 기반 스케줄링 기법은 가중치(weight) 기반 방식 [14][15][16] 과 속도(rate) 기반 방식 [7][17][18][19]으로 나뉘어 진다. 코어간 공정할당 스케줄링을 위해서 가중치 기반 방식은 부하(load)의 개념을 사용하고, 속도 기반 방식은 페이스(pace)의 개념을 사용한다. 코어의 부하는 간단히 말해서, 실행 가능한 태스크들의 가중치 합을 나타내고, 코어의 페이스는 지금까지 수행한 라운드(round)를 뜻한다. 단일 코

어 별 스케줄링은 가중치 기반과 속도 기반 방식 모두 태스크의 가중치를 사용한다.

대칭 멀티코어 시스템은 공정할당 스케줄링을 수행 시 단순히 CPU 시간만 고려하면 문제가 없다. 하지만 비대칭 멀티코어 시스템은 코어의 성능 비대칭성을 반드시 고려해야 한다. 비대칭 멀티코어 시스템의 공정할당 스케줄링 알고리즘은 (1) 정확하게 코어간 성능 비대칭성을 정량적으로 나타내는 방식과 (2) 나타내지 않는 방식으로 나눌 수 있다.

Li는 AMPS(Asymmetric Multiprocessor Scheduler)를 제안하였다 [20]. 이 연구에서 공정성(Fairness)은 모든 코어들의 스케일된 부하를 주기적으로 분산시킴으로써 얻어진다. 코어의 실행 큐에 존재하는 태스크 수를 코어의 컴퓨팅 능력으로 나눈 값을 스케일된 부하로 정의하였다. 이 때, 컴퓨팅 능력은 $F \times S$로 정의되며, F는 코어의 동작 주파수이고 S는 offline 벤치마크 테스트로 구해진 스케일링 값이다. 코어의 컴퓨팅 능력은 실행 중에 스케줄러에게 알려진다. AMPS는 몇 가지 단점이 있다. AMPS는 코어의 컴퓨팅 능력을 고정된 상수 값으로 표현하였다. 하지만 실제 운용 중에는 코어의 상태에 따라서 컴퓨팅 능력은 동적으로 다양하게 변한다. 더욱더, AMPS는 태스크들의 가중치를 고려하지 않아서 공정성(fairness) 본연의 의미를 지키지 못한다.

Li는 또한 A-DWRR(Asymmetric-Distributed Weight Round-Robin)을 제안하였다 [1]. 이는 DWRR의 확장된 버전이다. DWRR에서 제안한 태스크의 공정성은 round 즉, 속도의 개념으로 나타내었다 [7]. 한 round는 시스템에서 실행 가능한 모든 태스크들의 time slice들의 합으로 나타내었다. DWRR은 코어간 태스크 이동을 수행한다. 이러한 이동의 목적은 모든 태스크들의 round 값들이 같게 하기 위함이다. DWRR은 round 단위로 동작하기에 아주 정밀한 멀티코어에서의 공

정성을 보장하지는 않는다. 불행히도, A-DWRR은 이 특징을 그대로 가지고 있다. 코어간 비대칭성을 나타내기 위해서 A-DWRR은 상대적인 CPU 시간을 사용한다. 상대적인 CPU 시간은 CPU rating 값으로 스케일된 CPU 시간을 말한다. CPU rating이란 어떤 코어와 가장 느린 코어의 성능 차이를 숫자로 나타낸 값을 의미한다. 공정할당을 위하여 A-DWRR은 각 태스크들이 그들의 가중치에 비례하여 스케일된 CPU 시간을 갖도록 한다. A-DWRR은 코어의 CPU rating을 고정된 상수로 나타내었다. 따라서 DVFS에 따른 코어의 동작 주파수 변화는 CPU rating에 영향을 주지 못하는 단점이 있다.

Craeynest는 equal-progress 스케줄러를 제안하였다 [2]. 이 스케줄러는 공정할당을 위해서 모든 태스크들의 slowdown을 비슷하게 유지한다. 태스크의 slowdown은 실제 수행 시간과 고성능 코어 단독에서 수행했다고 가정한 시간의 비율을 나타낸다. 실제 환경에서 태스크는 고성능 코어와 에너지 효율적인 코어를 옮겨 다니며 수행된다. 따라서 어떤 태스크의 고성능 코어에서 단독으로 수행했다고 가정한 시간을 구하기 위해서는 에너지 효율적인 코어에서 수행한 시간들을 고성능 코어에서의 수행시간으로 변환해야 한다. 이를 위하여, 어떤 태스크가 수행될 때 사용한 두 가지 코어타입간의 CPI(cycle per instruction) 비율을 사용하여 변환한다. Equal-progress 스케줄러는 slowdown 값이 큰 태스크들을 고성능 코어에 할당하면서 공정할당 스케줄링을 수행한다. 하지만 이 스케줄러 역시 AMPS와 마찬가지로 태스크들의 가중치를 고려하지 않았다 [20].

[1][2][20]과 달리 비대칭성을 어떤 값으로 표현하지 않은 스케줄링 방식은 묵시적인 방법으로 성능 비대칭성을 고려하였다. Kazempour는 AASH(Asymmetry Aware Scheduler for Hypervisors)를 제안하였다

[21]. AASH는 hypervisor에서 수행되는 태스크들에 대한 공정할당 스케줄링이다. 이 스케줄러는 모든 테스크들이 고성능 코어에서 수행한 cycle 수와 에너지 효율적인 코어에서 수행한 cycle 수를 같게 유지한다. 이를 위하여 Xen hypervisor의 credit 스케줄러를 수정하였다. AASH는 주기적으로 어떤 태스크들이 고성능 코어에서의 cycle 수가 지나치게 높은지 체크한 후, 이들을 에너지 효율적인 코어로 이동시킨다.

R%-fair 스케줄러는 공정할당과 시스템의 처리량(throughput)을 동시에 고려한 스케줄러이다 [22]. 이 스케줄러는 고정된 분량(R%)의 고성능 코어에서의 cycle 수를 모든 태스크들이 균등하게 갖도록 한다. 나머지 분량((100-R)%)의 고성능 코어 cycle 수는 높은 처리량(throughput)을 필요로 하는 태스크들에게 할당된다. AASH와 마찬가지로 Xen hypervisor의 credit 스케줄러를 수정하였다.

AASH, R%-fair [21][22] 두 스케줄러는 시스템의 모든 태스크들이 같은 고성능 코어 cycle 수와 에너지 효율적인 코어 cycle 수를 갖도록 만드는 기법을 사용하였다. 이들은 본 연구에서 제안하는 스케줄링 알고리즘과 다른 기법이며, CPU 시간 할당에 있어서 태스크들의 가중치를 사용하지 않은 점을 생각하면 본연의 공정할당의 의미와는 거리가 먼 스케줄링 기법이다.

제 2 절 배경 지식

본 절에서는 본 학위논문의 나머지 부분에 대한 이해를 돕기 위하여 대상 시스템에 대한 설명을 한다. 본 학위논문에서는 대상 시스템으로, 비대칭 멀티코어 아키텍처 중 대표적으로 많이 쓰이는 ARM사의 빅리틀 멀티코어 아키텍처를 선정하였다. 이 아키텍처는 스마트폰, 테블릿, DTV

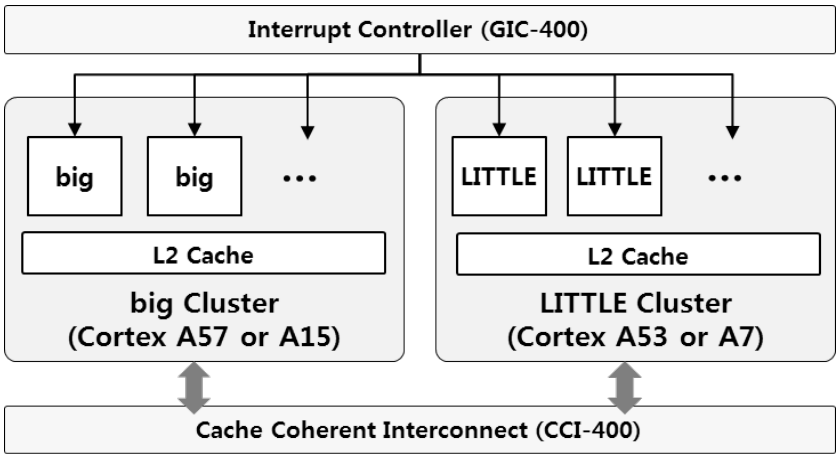


그림 2. 빅리틀 아키텍처

등의 상용 임베디드 시스템에서 저전력, 고성능을 목적으로 가장 많이 채택되는 아키텍처이다.

빅리틀 아키텍처는 두 개의 서로 다른 타입의 코어들로 이루어져 있다. 32 비트 시스템을 위하여 Cortex-A15는 빅코어, Cortex-A7은 리틀코어로 사용된다. 64 비트 시스템에서는 Cortex-A57과 Cortex-A53이 각각 빅코어와 리틀코어로 사용된다. 같은 타입의 코어들은 한 개의 클러스터에 속하게 된다. 이때, 빅코어들과 리틀코어들이 속한 클러스터는 각각 빅클러스터와 리틀클러스터이다. 그림 2는 클러스터 구조에 기반한 ARM사의 전형적인 빅리틀 아키텍처를 보여준다. 그림에서 CCI-400은 각 클러스터간의 공유된 cache 데이터들의 일관성을 보장하기 위하여 사용되는 하드웨어 장치이다. 두 개의 클러스터 혹은 각각의 빅리틀 코어는 GIC-400을 통해서 인터럽트 요청을 받게 된다 [5][6][27][37].

빅리틀 아키텍처에서 사용되는 비대칭 멀티코어 아키텍처의 코어 타입 선택 방식과 전체 코어 사용 방식은 각각 Switcher 모드와 GTS(Global

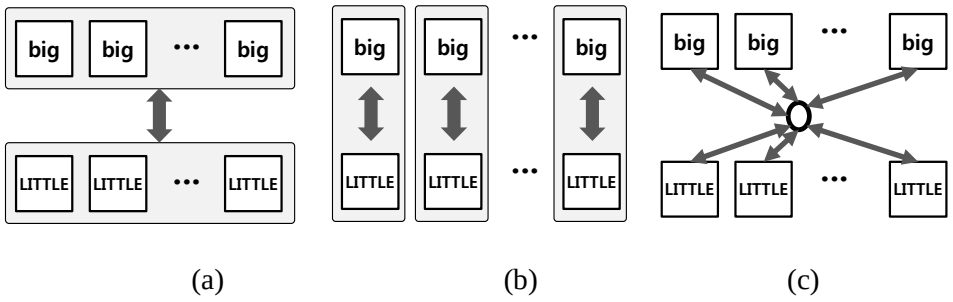


그림 3. 빅리틀 아키텍처의 동작 모드

Task Scheduling) 모드이다 [27]. 그림 3은 이러한 모드들을 보여주고 있다. 먼저 Switcher는 빅과 리틀의 두 가지 코어타입에서 어떤 코어를 사용할 지를 결정하는 Linux kernel의 소프트웨어 모듈이다 [38]. Switcher 모드는 클러스터간 이주 모드와 CPU간 이주 모드 두 가지 모드로 나누어지며, 그림 3의 (a)와 (b)가 각각 나타내고 있다. 클러스터간 이주 모드에서는 빅리틀 아키텍처가 실행 중에 빅클러스터와 리틀클러스터 중 한 개의 클러스터만 동작시키게 한다. CPU간 이주 모드에서는 한 개의 빅코어와 한 개의 리틀코어가 한 쌍을 이루며 이를 pair라고 부른다. 이 모드에서는, 실행 중에 하나의 pair에 속한 빅코어와 리틀코어 중 한 개의 코어만 동작시킨다.

그림 3의 (c)는 GTS 모드를 나타낸다. 이 모드에서는 모든 빅코어와 리틀코어는 각각 독립적으로 on/off 되며, 모든 코어들은 언제나 사용 가능하다. 따라서, 빅리틀 아키텍처의 GTS 모드는 그림 3에 나타난 세가지 운용모드 중 가장 좋은 성능과 스케줄링에 있어서 가장 좋은 유연성을 나타낸다.

이러한 하드웨어적 운용모드를 이용하여, 저전력이 요구되는 시스템에서는 Switcher 모드를 사용하고, 고성능이 필요한 시스템에서는 GTS

모드를 사용하도록 ARM사의 빅리틀 아키텍처는 지원하고 있다. 또한, Linaro 스케줄링 프레임웍은 ARM사의 빅리틀 아키텍처의 Switcher 모드와 GTS 모드에 필요한 각각의 소프트웨어 프레임웍을 제공하고 있다 [41]. 본 연구에서는 32 비트 아키텍처를 사용하여, 빅코어는 Cortex-A15를 나타내고, 리틀코어는 Cortex-A7을 사용하였다.

제 3 장 비대칭 멀티코어 아키텍처용 저전력 스케줄링

본 장에서는 비대칭 멀티코어 아키텍처에서 사용되는 최적화된 저전력 스케줄링 기법에 대하여 기술한다. 이를 위하여, 코어 타입 선택 방식인 Switcher 모드를 지원하는 ARM사의 빅리틀 아키텍처를 대상 시스템으로 정의한다. 그림 4는 빅코어인 Cortex-A15와 리틀코어인 Cortex-A7의 성능과 소비전력의 관계를 나타내고 있다. Switcher 모드로 동작 시, 그림에 나타낸 소비전력과 성능의 관계를 고려해서 코어의 상태를 빅코어 혹은 리틀코어로 제어한다.

Switcher 모드는 앞 장에서 설명한 바와 같이 클러스터간 이주 모드와 CPU간 이주 모드로 나누어진다. 클러스터간 이주 모드에서는 실행

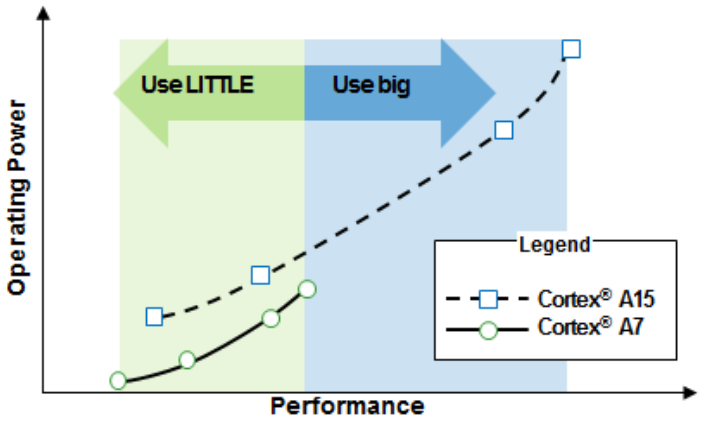


그림 4. 빅리틀 아키텍처의 성능과 소비전력의 관계

중에 항상 같은 타입의 코어들만 운용된다. 우선권은 항상 빅코어에 주어진다. 따라서, 시스템이 리틀코어로만 동작하는 중에, 리틀코어들 중 한 개의 코어만 빅코어를 필요로 한다고 하더라도, 모든 코어가 빅코어로 스위칭된다 [5][6].

클러스터간 이주 모드와 달리 CPU간 이주 모드에서는 각각의 빅리틀 pair 중 한 개의 코어가 사용되며, 다른 pair들의 상태와는 무관하게 동작한다. CPU간 이주 모드에서는 각 pair의 빅코어가 필요 없을 때는 off 시킨 후 리틀코어를 사용하므로 빅리틀 아키텍처에서의 에너지 효율성 극대화를 기대할 수 있다. 따라서 본 연구에서는 Switcher 모드 중 CPU간 이주 모드만 대상으로 하고, 이러한 하드웨어적인 특성에 최적화된 저전력 스케줄링 알고리즘을 제안한다.

제 1 절 시스템 정의

본 절에서는 빅리틀 아키텍처의 Switcher모드 중, 본 학위논문에서 대상으로 하는 CPU간 이주 모드가 사용하는 하드웨어적, 소프트웨어적 요소들에 대하여 설명한다.

1.1 가상 CPU와 부하분산

CPU간 이주 모드에서, 각 빅리틀 pair는 한 개의 가상 CPU로 나타내어진다 [38]. Linaro 스케줄링 프레임웍은 Linux CFS [39]를 사용하여 태스크들을 가상 CPU들에 할당하면서 스케줄링을 수행한다. 본 연구에서는 이러한 가상 CPU들의 집합을 S 라 정의하고 다음과 같이 나타낸다.

$$S = \{vCPU_0, vCPU_1, vCPU_2, vCPU_3\} \quad (1)$$

그림 5는 이에 대한 동작 예시를 나타낸다. 그림의 빗금 친 부분은 power-off된 상태의 코어를 나타낸다. 그림에서 알 수 있듯이 pair로 이루어진 가상 CPU 중 한 개의 코어만 동작하며, 각각의 가상 CPU들은 다른 가상 CPU의 상태에 상관없이 자신의 코어 타입을 결정한다.

Linux CFS는 집합 S에 속해있는 가상 CPU들 사이에서 태스크들을 옮긴다. 이때, 가상 CPU들 사이에서 직접 옮기기도 하고, 때로는 시스템의 준비 큐(wait-queue)에 태스크를 넣었다가 다시 꺼내어 가상 CPU들의 실행 큐(run-queue)에 할당하기도 한다. Linux CFS는 이러한 부하분산을 수행 시에 가상 CPU의 물리적 상태 즉, 빅코어인지 리틀코어인지, 혹은 가상 CPU의 동작주파수는 얼마인지 상관하지 않는다.

시스템의 준비 큐는 IO 동작으로 인하여 블로킹되거나 Sleep 상태로 진입한 태스크들을 가지고 있다. 실행 큐 q_s는 가상 CPU $s \in S$에서 수행할 태스크들의 집합을 나타낸다. Linux CFS는 가상 CPU들의 부하(load)를 고려하여 태스크들을 할당하고, 각 가상 CPU들간의 부하를 동일하게

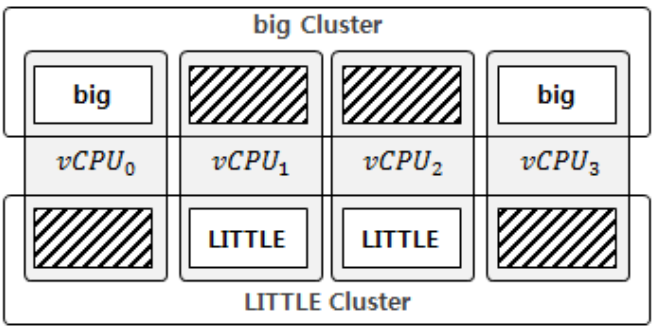


그림 5. CPU간 이주 모드 예시

맞추려고 노력한다. 이때 가상 CPU $s \in S$의 부하는 다음과 같이 정의된다.

$$L_s = \frac{\sum_{\tau \in T_k} w(\tau)}{\Gamma_s} \quad (2)$$

T_s는 q_s에 속해있는 실행 가능한 전체 태스크 집합을 나타내며, $w(\tau)$는 태스크 τ의 가중치를 나타낸다. Γ_s는 q_s가 할당된 가상 CPU s의 컴퓨팅 능력을 나타내는 상수이다. 이는 시스템의 아키텍처 특성에 따라서 미리 정해진 값이다. 이러한 방법으로 실행 큐에 할당되는 부하는 그 실행 큐가 할당된 가상 CPU의 현재 컴퓨팅 능력에 비례하여 할당된다.

실행 큐간의 적절한 부하분산을 위하여 두 가지 서로 다른 부하분산 정책이 사용된다. 첫째, 주기적 부하분산 정책이 있으며, 둘째, 유휴상태 부하분산 정책이 있다. 주기적 부하분산 정책은 주기적으로 각각의 실행 큐들의 부하를 살피고 그 차이가 어떤 불균형 상한 값을 넘는지를 체크한다. 이 때, 불균형 상한 값은 가장 큰 부하 값에서 현재 가상 CPU의 실행 큐의 부하 값을 뺌으로써 구해진다. 또한 이때 옮겨지는 태스크의 양은 불균형 정도에 따라 정해진다.

어떤 가상 CPU의 현재 수행되는 태스크가 할당 받은 time slice를 다 소진할 때까지 실행 큐에 다른 태스크들이 존재하지 않는다면 Linux CFS는 유휴상태 부하분산을 수행한다. 이 시점에서, 가장 부하 값이 큰 실행 큐로부터, 태스크들을 비어있는 실행 큐로 옮긴다.

1.2 Governor와 코어의 사용률

어떤 가상 CPU의 동작 주파수와 사용률은 현재 그 코어 위에서 수행되는 태스크들에 따라서 정해진다. Linux의 DVFS 정책은 코어의 사용률에 따라서 주파수를 변화시킨다. 높은 성능이 요구될 때는 코어의 사용률이 높아지며, 이에 반응하여 DVFS 정책은 governor를 통해서 코어의 주파수를 상승시킨다. 반대의 경우 소비전력을 최소화하기 위해서 코어의 주파수를 최소로 줄인다. 가상 CPU $s \in S$의 사용률은 다음과 같이 정의된다.

$$U(s) = 100 \times \frac{(Period - Idle\ time)}{Period} \tag{3}$$

위 식에서 $Period$ 는 연속된 governor epoch의 시간 축 상의 간격을 나타내며, 이 때 governor epoch는 ONDEMAND governor가 코어의 사

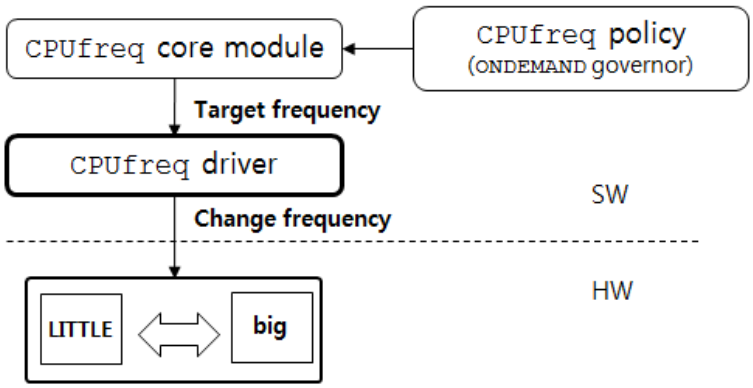


그림 6. Governor의 주파수 및 코어 타입 변경 동작 예시

용률을 관찰하기 시작한 시간이다 [36]. ONDEMAND governor는 Linux kernel에 구현되어 있다. 이는 코어의 사용률에 따라서 주파수를 조절하는 모듈이다.

그림 6은 governor가 실제 하드웨어를 제어해서 주파수를 변화시키거나 코어의 타입을 변경시키는 동작을 설명하고 있다. 그림에서 알 수 있듯이, governor에 의하여 정책이 결정되면 CPUfreq 모듈에 그 시그널이 입력된다. CPUfreq 모듈은 DVFS 정책에 정의된 주파수 값을 참고하여 목표로 하는 주파수 값을 결정한다. 이 결정된 값은 드라이버에 전달되고, 드라이버는 직접 코어를 제어하여 주파수를 변경시킨다. 이때 전달되는 주파수 값과 코어의 현재 상태에 의하여 코어 타입이 변화되며, 이에 대한 설명은 다음 세부 절에 나타낸 상태도를 통하여 자세히 설명한다.

ONDEMAND governor 모듈은 코어의 사용률을 주기적으로 관찰한다. 이 때 사용률의 상한 값과 하한 값을 사용한다. Governor는 한 관찰 구간 사이에서 관찰된 코어의 사용률이 상한 값(80%)를 넘을 경우, 가상 CPU가 취할 수 있는 가장 높은 주파수 값으로 현재의 주파수를 변경한다. 반대로 하한 값(70%)보다 낮을 경우 현재의 주파수에서 한 단계 낮은 주파수로 변경한다.

1.3 CPU간 이주 모드에서의 DVFS 모델

빅리틀 아키텍처에 사용되는 ONDEMAND governor 동작을 나타내기 위해서, 우리는 단순화된 DVFS 모델을 사용한다. 이모델을 위하여, 코어의 동작 주파수는 두 가지 상태로 정의한다. {low, high}의 두 가지 동작 주파수를 나타내는 상태는 줄여서 {L, H}로 표시한다. 다음으로, 코어의 사용률은 세가지 상태로 정의한다. {low, medium, high}의 세가지 사용

률을 나타내는 상태는 줄여서 {L, M, H}로 표시한다. 따라서 Cortex-A15/Cortex-A7로 이루어진 pair는 6가지 상태 중 한 개로 표현된다. 이는 (주파수, 사용률)로 표기되며 모두 열거하면 다음과 같다: (L, L), (L, M), (L, H), (H, L), (H, M), (H, H). 그림 7은 이들의 상태 변이를 나타낸 상태도 이다. 가로 축은 {L, H} 값으로 표현되는 코어의 동작 주파수를 나타내고, 세로 축은 {L, M, H} 값으로 표현되는 코어의 사용률을 나타낸다.

실제 운용 중에는, 코어가 취할 수 있는 많은 종류의 주파수가 존재한다. 또한, % 값으로 측정되는 다양한 값의 사용률이 있다. 이러한 주파수와 사용률 값을 간단하게 {L, H}와 {L, M, H}로 표현하고자, 표 1과 표 2에 간략히 정리된 이들의 정의를 나타내었다.

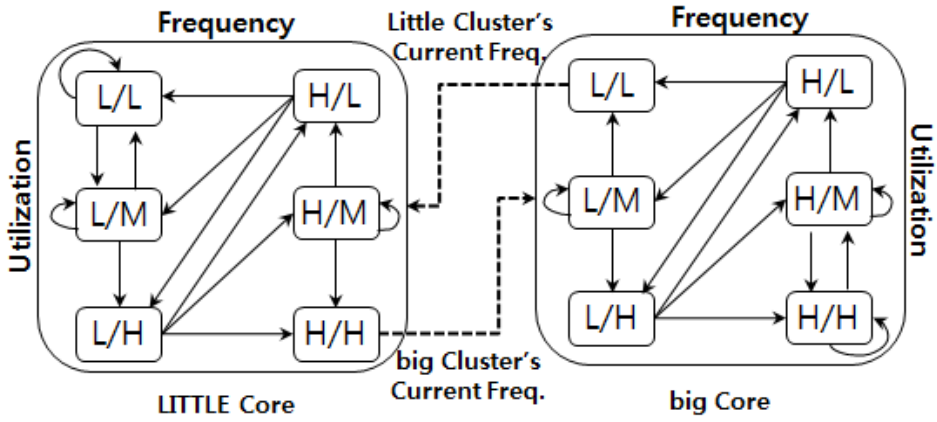


그림 7. 가상 CPU의 상태도

표 1. 가상 CPU의 주파수 상태의 의미

상태	L	H
주파수(f)	f < 최대 허용 주파수	f = 최대 허용 주파수

표 2. 가상 CPU의 사용률 상태의 의미

상태	L	M	H
사용률(U)	U<하한 값	하한값 <U <상한값	U>상한 값

그림 7에서 보여주듯이, 리틀코어가 (H, H) 상태에 있을 때만 다음 상태에서 빅코어로 스위칭되는 것을 알 수 있다. 새롭게 활성화된 빅코어의 주파수는 빅클러스터의 현재 주파수로 결정된다. 그리고 새롭게 활성화된 빅코어의 사용률은 빅클러스터의 주파수와 할당된 태스크들에 따라서 결정된다.

반대로, 빅코어가 (L, L) 상태에 있을 때만 다음 상태에서 리틀코어로 스위칭된다. 그리고 이때, 빅코어의 실행 큐에 있던 태스크들은 새롭게 활성화된 리틀코어에 할당된다. 이는 pair로 이루어진 가상 CPU는 한 개의 실행 큐가 있고, 같은 pair 내부의 빅코어와 리틀코어는 이 실행 큐를 공유하기 때문이다. 새롭게 활성화된 리틀코어의 주파수는 현재 리틀 클러스터에서 사용되는 주파수로 설정되며, 이 주파수와 할당된 태스크들에 따라서 사용률이 결정된다.

그림 8은 가상 CPU $s \in S$가 시간에 따라서 상태가 변경되는 모습을 예를 들어서 설명하고 있다. 각 governor epoch들은 점선으로 이루어진 화살표로 나타내었다. 그리고 이들의 시간 축 상의 표기는$GE_{(s,x)}$로 하였

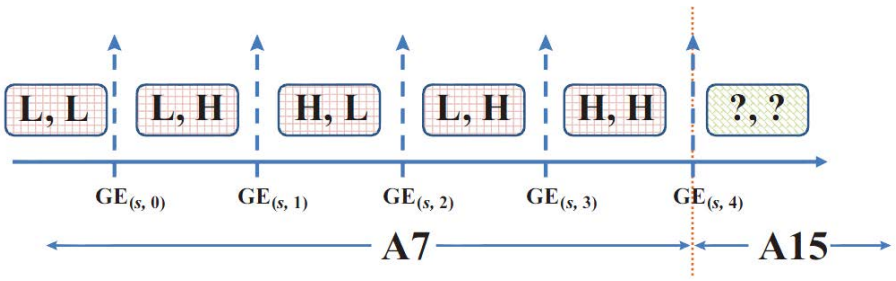


그림 8. 가상 CPU의 상태 변화 예시

으며, x는 governor epoch의 순서를 나타낸다. ONDEMAND governor는 각 governor epoch에서, 현 시점과 바로 직전의 governor epoch 사이의 코어의 사용률을 측정한다. 측정된 값에 따라서, governor는 가상 CPU의 주파수를 조절하거나 pair 안에서 코어간 스위칭을 수행한다.

그림 8의 예를 들어 설명하면 다음과 같다. 가상 CPU의 상태가 리틀 코어에서 시작한다면, $GE_{(s,1)}$에서 governor는 $GE_{(s,0)}$과 $GE_{(s,1)}$사이에서 코어 사용률이 'H'임을 착안하고 $GE_{(s,1)}$과 $GE_{(s,2)}$사이의 코어의 동작 주파수를 'H'상태로 변경한다. $GE_{(s,2)}$에서는 $GE_{(s,2)}$과 $GE_{(s,1)}$사이에서 코어의 사용률이 'L' 상태임을 착안하고 동작 주파수를 한 스텝 낮은 주파수로 설정한다. 따라서 주파수는 'L' 상태로 변경된다. 코어의 사용률은 $GE_{(s,2)}$과 $GE_{(s,3)}$사이에서 다시 높아짐에 따라서 governor는 동작 주파수를 상승시킨다. 마침내 $GE_{(s,3)}$과 $GE_{(s,4)}$사이에서 주파수와 사용률이 (H, H) 상태가 되고, $GE_{(s,4)}$에서 빅코어로 스위칭된다. 코어의 상태 즉 (주파수, 사용률)은 실행 중에 결정된다. 따라서 $GE_{(s,4)}$이후 상태는 (?, ?)로 표시하였다. 할당된 주파수는 빅클러스터의 주파수로 할당된다. 이때 모든 빅코어가 off 상태였다면, DVFS 정책에 의하여 빅코어의 가장 낮은 주파수로 설정된다

제 2 절 문제 정의

Linaro 스케줄링 프레임웍에서 사용하는 Linux CFS는 코어의 사용률을 고려하여 부하분산을 수행하지 않는다. 이로 인하여, 어떤 태스크가 필요 없이 높은 주파수로 동작하는 코어에 할당하거나 혹은 필요 없이 어떤 코어의 주파수를 상승시킬 수 있다. 또한, 불필요한 리틀코어에서 빅코어로 스위칭을 야기할 수 있다. 이러한 현상은 모두 코어들이 필요 이상의 에너지를 사용하도록 만드는 원인을 제공한다.

예를 들면, 두 개의 가상 CPU $s_1, s_2 \in S$가 있고, s_1, s_2 모두 현재 리틀 코어 상태이며 모두 마지막 governor epoch 시점에서 (L, L) 상태라고 하자. 그리고 다음의 식을 만족한다고 가정한다.

$$\mathrm{L}_{s_1} < \mathrm{L}_{s_2} ,$$

$$U(s_1) > U(s_2)$$

Linux CFS에게는, 시스템의 준비 큐에서 빠져 나온 새로운 한 태스크가 사용률이 높은 s_1의 실행 큐에 할당하는 것이 타당하게 보인다. 그 이유는 s_1의 시행 큐 부하가 s_2의 실행 큐 부하보다 작기 때문이다.

하지만, s_1에 태스크를 할당하는 것은 (L, H) 상태로 코어의 상태가 바뀌는 확률을 높이게 된다. 이는 governor가 다음 governor epoch 이후의 동작 주파수를 증가시켜 에너지 사용을 증가시키게 된다.

제 3 절 해결책

본 절은 빅리틀 아키텍처의 CPU간 이주 모드에 적합한 사용률인지 기반 부하분산 알고리즘(Utilization-aware Load Balancing)을 제안한다. 이 알고리즘은 추정기(estimator)를 사용한다. 어떤 태스크가 시스템의 준비 큐에서 빠져 나와 실행 가능한 상태가 되면 새로 진입하게 될 실행 큐를 선정하게 된다. 이때, 추정기의 역할은 어떤 실행 큐로 할당을 해야 에너지 사용에 가장 적게 영향을 미치는지 계산한다. 본 연구에서는 이 시점을 부하분산을 수행하는 시점으로 정의한다. 또한, 두 가지 추정기를 사용하여, 알고리즘에 각각 적용하고 그 실효성을 검증하였다.

하기 절의 3.1은 사용률인지 기반 부하분산 알고리즘의 전체적인 설명을 하고 3.2와 3.3은 알고리즘에 사용된 두 가지 추정기에 대한 내용을 기술하였다.

3.1 사용률인지 기반 부하분산 알고리즘

ALGORITHM 1은 본 연구에서 제안하는 사용률인지 기반 부하분산 알고리즘(Utilization-aware load balancing)의 의사코드를 나타낸다. 이 알고리즘은 두 단계의 스텝으로 이루어져있다. 첫째, 라인 5~7에 나타난 것처럼, 가장 부하가 큰 실행 큐로부터 태스크를 빼내어 가장 부하가 작은 실행 큐로 옮긴다. 이는 CFS의 기본 철학인 가중치 기반 부하분산 정책과 동일한 방식이다. 본 연구에서는 Linux CFS의 주기적 부하분산 정책과 유휴상태 부하분산 정책을 비활성화 시켰다. 따라서 이 부분이, 시스템이 가상 CPU간 부하분산을 수행하는 유일한 시점이 된다.

둘째, 시스템의 준비 큐에서 빠져 나온 태스크를 할당할 새로운 실행 큐를 선정한다. 이때, 추정기를 사용하며, 그림 7에 표현된 상태도의 동작을 고려한다. 그리고 주파수 상승이나 빅코어로 스위칭될 확률이 가장

ALGORITHM 1. UTILIZATION-AWARE LOAD BALANCING

Input: $S = \{vCPU_0, vCPU_1, vCPU_2, vCPU_3\}$

1: Imbalance= 1.25

2: τ = Dequeue(q_{wait})

3: CPU_{idlest} = Argument $s \in S$ of the minimum L_s

4: $CPU_{busiest}$ = Argument $s \in S$ of the maximum L_s

5: **While** ($L_{CPU_{busiest}} > L_{CPU_{idlest}}$ * Imbalance) **do**

6: $Task_{popped}$ = Dequeue($q_{CPU_{busiest}}$)

7: Enqueue($q_{CPU_{idlest}}$, $Task_{popped}$)

8: **END While**

9: CPU_{Util} = Argument $s \in S$ of the minimum $E[U(s,\tau)]$

10: Enqueue ($\boldsymbol{q_{CPU_{Util}}}$, $\boldsymbol{\tau}$)

작은 실행 큐를 찾는다. $E[U(s,\tau)]$는 다음 governor epoch 이후에 태스크 τ가 삽입될 때 코어의 사용률에 대한 예측 값을 나타낸다. 따라서 이 알고리즘은 이 값이 가장 작은 코어의 실행 큐를 결과값으로 제시한다.

그림 9.는 이 추정기가 동작할 때 사용되는 시점을 나타낸 그림이다. τ가 가상 CPU s로 할당될 때, $\Delta T_{(s,k)}$는 k번째 governor epoch인 $GE_{(s,k)}$부터 가상 CPU s가 선정되는 시점인 T_{vCPU}까지의 시간이다. T_{vCPU}는 절대적인 시간이며, 모든 가상 CPU에게 동일하게 발생한다. i와 j가 다를 때, $GE_{(i,k)}$ 는 반드시 $GE_{(j,k)}$와 같지는 않다. 그리고, i와 j가 다를 때, $\Delta T_{(i,k)}$가 $\Delta T_{(j,k)}$와 같지는 않다. 즉, 각 가상 CPU 마다 서로 다른 시점의 governor epoch가 존재한다. $U(s)$는 가상 CPU s의 $[GE_{(s,k)}, T_{vCPU}]$ 동안

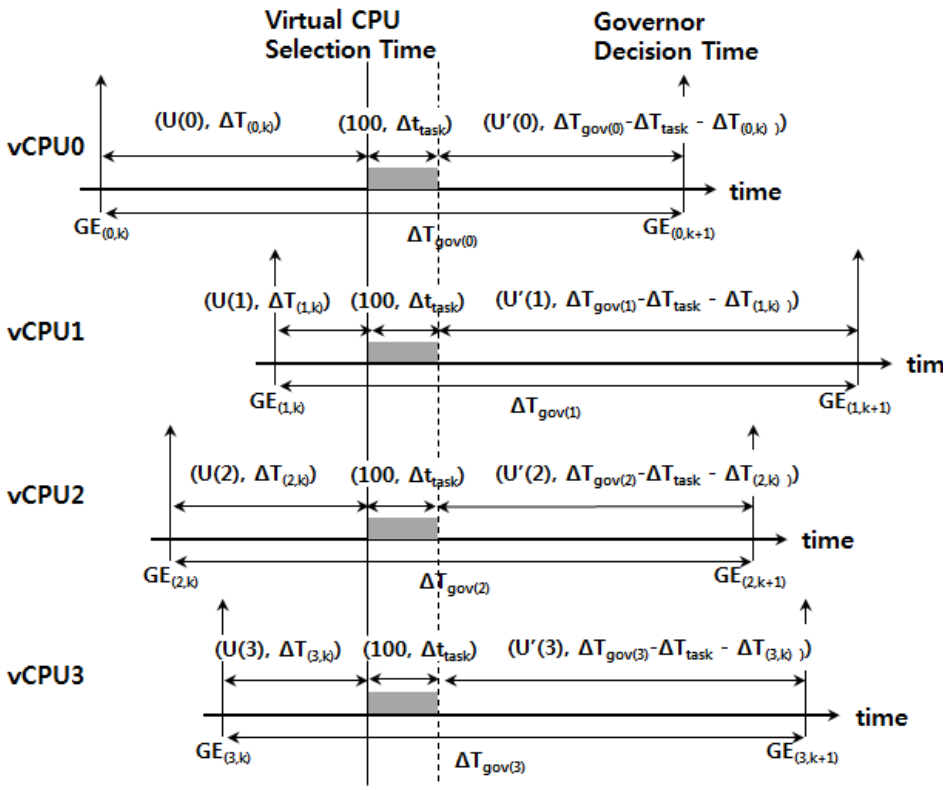


그림 9. 가상 CPU 선택 시점과 Governor Epoch의 관계

측정된 사용률이다. $\Delta T_{gov(s)}$는 가상 CPU s의 governor epoch의 간격을 나타낸다 그리고 $\Delta T_{gov(i)} \approx \Delta T_{gov(j)}$ 을 만족한다. 모든 i 와 j에 대하여, $U'(s)$는 τ가 가상 CPU s로 할당된 후 추정되는 사용률을 나타낸다.

3.2 사용률 기반 추정기

본 연구에서 제안하는 첫 번째 추정기는 다음 사항을 가정한다.

$E_A[U(s,\tau)]$는 첫 번째 추정기가 예측하는 τ가 가상 CPU s로 할당될 때, 다음 governor epoch 이후의 코어 사용률이다. 이 추정기는 τ가 가상 CPU s의 실행 큐에 할당되어도 할당되기 전의 사용률을 그대로 유지할 것이라는 점을 이용한다. 이는 $U'(s) = U(s)$를 뜻한다. 이를 식으로 표현하면 다음과 같다.

$$E_A[U(s,\tau)] = U(s)$$

3.3 수행이력을 반영한 사용률 기반 추정기

두 번째 추정기는 $E_B[U(s,\tau)]$값을 사용하며, 이는τ 가 가상 CPU s로 할당될 때, 예상되는 다음 governor epoch 이후의 코어 사용률이다. 첫 번째 추정기와 달리, 이 추정기는 그림 9에서 $[GE_{(s,k)},\ GE_{(s,k)} + \Delta T_{(s,k)} + \Delta T_{task}]$ 동안의 사용률을 계산한다. 즉, 다시 말하면, T_{vCPU} 시점에 태스크 τ가 가상 CPU s에서 바로 수행되는 것을 가정한다. 그 결과, τ의 수행 시간에 따른 코어 사용률 변동이 일어남을 반영하여 $E_B[U(s,\tau)]$값을 구한다.

이를 위하여, 태스크 τ의 수행시간 정보가 필요하다. 본 연구에서는 이 수행시간 정보를 구함에 있어서, 최근 태스크 τ가 수행했던 16개의 수행시간을 기록하고 이들의 평균을 구한다. 이 값을 ΔT_{sma}라고 표현하고, 이 표현에서 sma는 'Simple Moving Average'의 줄임 말이다. 따라서 $\Delta T_{task} = \Delta T_{sma}$를 뜻하게 된다. τ는 최근 16번 수행될 때, 집합 S에 속한 모든 가상 CPU에서 수행될 수 있기에 ΔT_{sma}는 가상 CPU와 무관하게 단순히 'Simple Moving Average' 방법을 이용하여 구한다. 본 연구에서 제안하는 수행이력을 반영한 사용률 기반 추정기의 결과 값은 다음과 같

다:

$$E_B[U(s,\tau)] = U'(s) = \ U(s) \ \times \frac{\Delta T_{(s,k)}}{\Delta T_{(s,k)} + \Delta T_{sma}} + \frac{100 \times \Delta T_{sma}}{\Delta T_{(s,k)} + \Delta T_{sma}}$$

제 4 장 비대칭 멀티코어 아키텍처용 공정할당 스케줄링

본 장에서는 비대칭 멀티코어 아키텍처에서 사용되는 최적화된 공정할당 스케줄링 기법에 대하여 기술한다. 이를 위하여, 전체 코어 사용 방식인 GTS(Global Task Scheduling) 모드를 지원하는 ARM사의 빅리틀 아키텍처를 대상 시스템으로 정의한다. GTS 모드는 Switcher 모드와 달리 시스템의 모든 빅코어와 리틀코어를 동시에 사용할 수 있다. 따라서 가장 높은 성능을 얻을 수 있어서 개발자들에게 가장 선호되는 빅리틀 시스템의 동작 모드이다.

Linaro는 Linux CFS를 확장하여 GTS모드에 적합한 스케줄링 프레임웍을 제공하고 있다. Linux CFS는 태스크들의 상대적 진척 정도를 virtual runtime로 정의한다. 그리고 태스크들의 virtual runtime을 서로 같게 유지하려고 노력하면서 공정할당 스케줄링을 수행한다. 하지만 공정할당 측면에서 봤을 때, Linaro 스케줄링 프레임웍은 두 가지 문제점이 있다.

첫째, CFS는 단순히 태스크의 가중치에 비례하여 per-core 스케줄링을 수행한다. 어떤 동일한 태스크 τ_i와 τ_j가 각각 빅코어와 리틀코어에서 수행된다고 하면, τ_i의 완료시간이 τ_j보다 당연히 빠를 것이다. 이는 virtual runtime을 통하여 태스크들의 상대적 진척도를 비슷하게 맞추는 CFS의 기본 취지에도 어긋나게 된다. 또한, 같은 코어타입에서 두 동일한 태스크가 수행되더라도, 동작주파수가 큰 코어에서 수행한 태스크가

더 빨리 끝날 것이다. 따라서 비대칭 멀티코어 시스템에서는, 어떤 태스크의 CPU 시간을 계산 시 그 태스크가 사용했던 코어타입과 동작 주파수를 고려해야 한다. 따라서 태스크의 CPU 시간은 코어타입과 동작 주파수에 의하여, 스케일된 CPU 시간으로 계산되어야 한다.

태스크의 스케일된 CPU 시간을 계산 시, 코어의 상태뿐만 아니라 태스크별 수행 특성 역시 고려해야 한다. 이를 예를 들어 설명하면 다음과 같다. 어떤 동일한 태스크 τ_i와 τ_j가 각각 높은 주파수의 빅코어와 낮은 주파수의 리틀코어에서 수행된다고 가정한다. 두 동일한 태스크의 동작 특성은 높은 branch-prediction miss와 큰 메모리 stall을 가지고 있다고 하자.

이러한 경우 τ_i는 빅코어와 높은 주파수의 장점을 다 활용할 수 없다. 하지만, 앞서 설명한 것처럼 스케일된 CPU 시간을 계산시, 코어타입과 동작 주파수만 고려하면, 태스크 τ_i와 τ_j가 진척 정도가 유사함에도 불구하고 태스크 τ_i가 큰 스케일된 CPU 시간을 기록하게 될 것이다.

이 모두를 종합적으로 정리하면 다음과 같은 결과를 나타낸다. 태스크의 스케일된 CPU 시간 산정은 (1)코어타입, (2)동작 주파수, (3)태스크의 동작 특성 등을 모두 고려해야 하고 이들을 per-core 스케줄링에 반영해야 한다.

둘째, CFS는 태스크들의 상대적 진척 정도를 per-core 기반으로 유지하고, 이를 전체 코어로 확대하지 못한다. CFS는 코어 전체에 걸쳐, 모든 태스크에 대하여 virtual runtime을 동일하게 유지하려고 한다. 이를 위해서 코어간 부하분산을 수행한다. 이때 사용하는 부하분산 정책은 태스크의 가중치에 기반을 두고 있다. 즉, 코어당 할당된 태스크들의 가중치 합을 동일하게 유지시키기 위하여 코어간 부하분산을 수행한다. 하지만, 이는 태스크간 상대적 진척도인 virtual runtime을 동일하게 유지하는 것

에 실효성 있는 해결책을 주지 못한다.

본 장에서는 이러한 문제점들을 해결하는 공정할당 스케줄링 기법을 제안한다. 1절에서는 대상 시스템을 정의하고, 현재 사용되는 비대칭 멀티코어 아키텍처용 소프트웨어를 설명한다. 2절에서는 공정할당을 정형화해서 정의하고 3절에서는 해결하고자 하는 문제를 정의한다. 마지막으로 4장에서는 정형화된 문제의 해결책을 제시한다.

제 1 절 시스템 정의

본 절에서는 공정할당 스케줄링 알고리즘이 대상으로 하는 비대칭 멀티코어 아키텍처를 정의하고, 본 학위논문의 이후 부분에서 사용되는 수학적 기호 및 용어를 정리한다. 이어, 대상 시스템으로 사용하는 빅리틀 아키텍처의 GTS 모드를 위하여 개발된 소프트웨어 프레임웍을 설명한다.

1.1 대상 시스템 모델링 및 용어 정리

본 학위논문의 이후 기술된 내용을 이해하는 데 도움을 주기 위하여, 자주 사용되는 수학적 용어들의 의미를 분명하게 하고자 한다. 이들을 표 3에 정리하였다. 이들 수학적 용어들은 제안된 기법의 핵심 기능들을 정형화하여 표현하기 위해서 사용한다. 본 절에서는 공정할당 스케줄링의 대상 시스템인 ARM사의 빅리틀 아키텍처를 표 3에 나타낸 기호를 사용하여 모델링 한다.

그림 10은 본 연구의 대상 시스템을 간략하게 모델링한 그림이다. 대

상 시스템은 한 개의 빅클러스터와 한 개의 리틀클러스터로 이루어져 있고, 각 클러스터는 동일한 r 개의 빅코어들과 s 개의 동일한 리틀코어로 구성된다. 이를 나타내면, 빅클러스터는 $P_b = \{p_1^b, p_2^b, \dots, p_r^b\}$, 리틀클러스터는 $P_l = \{p_1^l, p_2^l, \dots, p_s^l\}$ 로 표현된다. 그리고 모든 코어들은 GTS 모드로 동작한다.

표 3. 수학적 기호 및 용어 설명

Symbol	Definition
r	Number of big cores in the system
s	Number of little cores in the system
$P_b = \{p_1^b, p_2^b, \dots, p_r^b\}$	Set of big cores
$P_l = \{p_1^l, p_2^l, \dots, p_s^l\}$	Set of little cores
$Q_b = \{q_1^b, q_2^b, \dots, q_r^b\}$	Set of run-queues for big cores
$Q_l = \{q_1^l, q_2^l, \dots, q_s^l\}$	Set of run-queues for little cores
n	Number of tasks in the system
$T = \{\tau_1, \tau_2, \dots, \tau_n\}$	Set of tasks in the system
$w(\tau_i)$	Weight of task τ_i
t	Wall clock time
λ	Balancing period
$R_i(t)$	relative performance of a task τ_i at time t
$\hat{c}_i(t)$	Scaled CPU time of task τ_i
$\hat{v}_i(t)$	SVR of τ_i at time t
$\lvert\hat{v}_{i,j}(t)\rvert$	SVR difference between τ_i and τ_j at time t
$\hat{v}_{max}(t)$	The biggest $\lvert\hat{v}_{i,j}(t)\rvert$ at time t
$\Delta\hat{v}_i(r)$	SVR increment of τ_i during r^{th} load balancing period $[(r-1)\lambda,\ r\lambda]$
$\Delta\hat{v}_{i,j}(r)$	Difference between $\Delta\hat{v}_i(r)$ and $\Delta\hat{v}_j(r)$
$w_{\max}/w_{\min}$	Maximum/minimum weight in the system

한 개의 빅코어 $p_i^b \in P_b$는 한 개의 실행 큐 q_i^b를 가진다. 이들 실행 큐들은 한 개의 집합으로 나타내면, $Q_b = \{q_1^b, q_2^b, \ldots, q_r^b\}$로 나타낼 수 있다. 마찬가지로, 리틀코어들은 실행 큐 집합인 $Q_l = \{q_1^l, q_2^l, \ldots, q_s^l\}$를 가진다. 또한, 빅코어와 리틀코어가 운용 중에 취할 수 있는 동작 주파수 집합을 각각 $F(P_b)$와 $F(P_l)$로 나타낸다.

이러한 클러스터 구조로 이루어진, 빅리틀 아키텍처 위에서 운용되는 태스크 모델은, n개의 태스크로 이루어진 하나의 집합으로 나타낸다. 그리고, 이 집합은 $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$로 표현된다. 어떤 태스크 $\tau_i \in T$의 가중치(weight)는 고정된 상수이며 $w(\tau_i)$로 나타낸다. 모든 집합 T에 속한 태스크들은 Linaro 스케줄링 프레임웍이 사용하는 Linux CFS에 의하여 스케줄링 된다. 따라서, 매 스케줄링 tick마다 per-core 스케줄링을 수

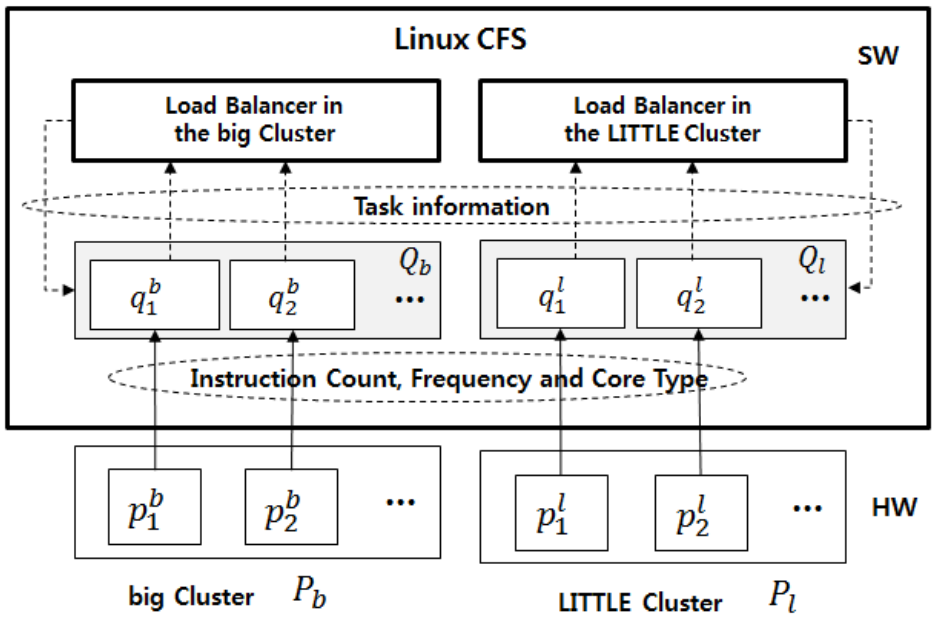


그림 10. 대상 시스템 모델링

행하며, 각 클러스터별로 가중치 기반 부하분산을 수행한다. 이때, 부하분산은 각 클러스터마다 독립적으로 이루어지며, 부하분산 주기는 λ이다.

성능 비대칭성을 스케줄링 알고리즘에 도입하기 위하여, 본 연구에서는 relative performance를 정의한다. 어떤 태스크 τ_i 의 relative performance는 리틀코어의 최소 주파수로 동작 시 측정된 성능과 현재 코어 상태에서 측정된 성능의 비율을 나타낸다. 이때 리틀코어의 최소 주파수는 f_{min}^{l}로 나타내며, $f_{min}^{l} \in F(P_l)$를 만족한다. 각 태스크의 현재 상태 성능은 런타임에 측정된, 스케줄링 tick 사이에 발생한 인스트럭션 수를 나타낸다. 이를 IPT(instruction counts per scheduling tick)로 표기한다. IPT에 대한 측정 방법과 자세한 설명은 본 장의 4절에서 다룬다. $R_i(t)$는 태스크 τ_i의 시간 t에 측정된 relative performance로 정의한다.

1.2 GTS 모드를 위한 Linaro 스케줄링 프레임웍

ARM사의 빅리틀 아키텍처는 GTS 모드를 지원하기 위한 소프트웨어 프레임웍으로써, Linaro 스케줄링 프레임웍을 사용한다. 이는 그림 11에 나타낸 것처럼, Linux의 CFS를 확장해서 빅리틀 아키텍처에 맞게 재구성한 소프트웨어이다.

비대칭 멀티코어 아키텍처가 주는 장점을 최대한 사용하기 위해서 Linaro 스케줄링 프레임웍은 클러스터간 태스크 이주 정책을 기존 Linux CFS에 추가하였다. 이 정책은 현재 CPU-intensive한 태스크들은 빅코어에서 수행되게 하고 백그라운드 태스크와 같은 나머지 태스크들은 리틀코어를 사용하도록 하는 기법이다. 이로써, 시스템의 성능을 향상시킬 수 있다.

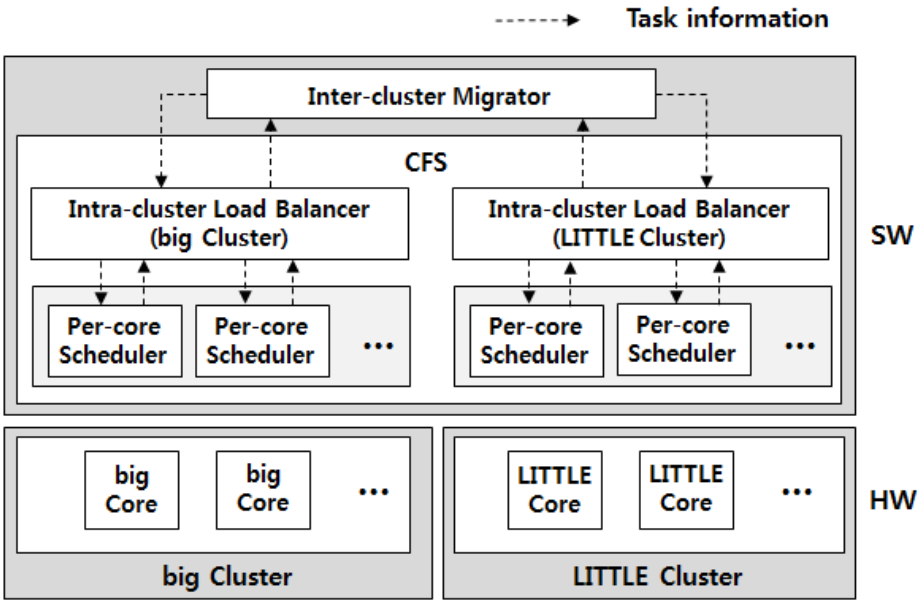


그림 11. Linaro 스케줄링 프레임웍

Linaro 스케줄링 프레임웍은 Linux CFS의 per-core 스케줄링과 가중치 기반 부하분산 정책을 그대로 사용한다. 이때, 부하분산 정책은 가중치를 기반으로 하고, 클러스터 내부에서만 수행된다. 클러스터간 부하분산은 앞서 설명한 클러스터간 태스크 이주정책을 사용한다.

■ **클러스터간 태스크 이주정책**

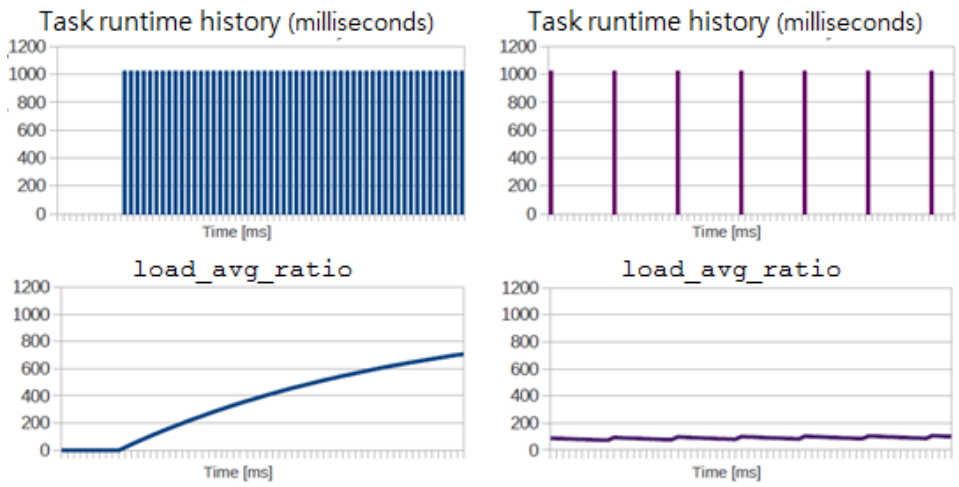
태스크들을 적절한 코어타입에 할당하기 위해서 Linaro 스케줄리 프레임웍은 빅클러스터 P_b 와 리틀클러스터 P_l 사이에 이주정책(Inter-cluster migrator) 을 사용한다. 이를 위하여 각 태스크 별 *load_avg_ratio* 을 주기적으로 체크하며 이는 다음과 같이 정의하고 있다:

$$load\_avg\_ratio = \frac{w_0}{runnable\_avg\_period} \times runanable\_avg\_sum$$

위 식에서 w_0 는 Linux kernel에서 정의하고 있는 nice 값 0 의 가중치를 나타낸다. *runnable_avg_period* 는 해당 태스크의 전체 life time을 나타내고, *runnable_avg_sum*은 태스크가 실행상태로 설정된 시간들을 최근 시간에 가중치를 두어 계산한 값을 나타낸다. 어떤 태스크의 *runnable_avg_sum*을 계산하기 위해서, 실행 가능한 상태로 되어졌던 시간들을 1ms 크기의 segment로 잘게 나눈다. 이 segment들이 최근 시간에 가까울수록, 큰 값을 갖게 된다 [42]. 따라서, 어떤 태스크가 최근에 실행 가능한 상태로 오랜 시간 존재했다면, 이 태스크의 *load_avg_ratio* 는 증가한다.

그림 12는 segment와 *load_avg_ratio*의 관계를 나타내고 있다. 그림의 (a)는 최근까지 태스크가 수행했던 시간이 촘촘하게 기록되고 있다. 따라서 그림에서 알 수 있듯이, *load_avg_ratio* 값은 증가하고 있다. 그림의 (b)는 최근까지 태스크가 수행했던 시간이 기록되었으나, 그 빈도가 드물게 기록되고 있다. 따라서 *load_avg_ratio* 값은 아주 소폭 증가되다가 다시 작은 값으로 돌아오는 과정을 반복하고 있음을 알 수 있다.

어떤 태스크가 리틀코어에서 동작하고 있을 때, 만일 *load_avg_ratio* 값이 어떤 경계값(up-threshold)보다 크게 되면, 이 태스크는 빅클러스터의 한 코어로 이주된다. 반대로 빅코어에서 동작할 때, *load_avg_ratio* 값이 어떤 경계값(down-threshold)보다 작게 되면 이 태스크는 리틀클러스터의 한 코어로 이주되게 된다.



(a) (b)

그림 12. Segment에 따른 ***load_avg_ratio*** 변화

시스템의 성능은 현재 동작중인 태스크의 동작 속도에 많은 영향을 받게 된다. 시스템 운용시간이 증가하면, 이 동작중인 태스크들은 큰 *load_avg_ratio* 값을 가지게 된다. 따라서 빅코어에 남아 있거나 혹은 리틀코어에서 빅코어로 옮겨진다. 반대로, 백그라운드 태스크처럼 최근 실행 가능한 상태로 있었던 시간이 작은 태스크들은 리틀코어에 남거나 혹은 빅코어에서 리틀코어로 옮겨지게 된다. 이러한 클러스터간 이주 정책으로 인하여 시스템의 성능은 향상되게 된다.

■ **클러스터 내부 부하분산 정책**

태스크들에게 할당될 코어 타입이 위에서 언급한 클러스터간 이주정책에 의하여 결정되면, 클러스터 내부에서의 부하분산 정책(Intra-cluster load balancer)에 의하여 태스크들은 코어에 할당된다.

클러스터 내부에서의 부하분산 정책은 3장 1절에서 설명한 부하분산 정책과 동일하다. 하지만, 식 (2)에 나타낸 Γ_s의 역할은 조금 다르다. Γ_s는 빅코어일 때의 값과 리틀코어일 때의 값 두 종류를 가지고 있다. CPU 간 이주 모드에서는, 서로 다른 Γ_s 값을 갖는 코어들 사이에서 부하분산이 이루어진다. 따라서 식 (2)를 계산할 때 Γ_s가 반드시 필요하다.

이와 다르게 GTS 모드용 Linaro 스케줄링 프레임웍은 동일 코어들로 이루어진 클러스터 내부에 한하여 부하분산을 수행한다. 클러스터 내부 코어들은 Γ_s 값이 모두 같기 때문에, 이 값은 부하분산 정책에 있어서 아무런 의미가 없다.

■ Per-core 스케줄링

클러스터내부의 부하분산정책에 의하여 태스크의 코어가 결정되면, Linux CFS는 per-core 스케줄링을 수행한다. CFS는 분산 실행 큐 알고리즘이며, 이는 각 코어에 지정된 실행 큐를 사용한다 [23]. 각 실행 큐는 virtual runtime이 작은 순서대로 정렬되어 있는 실행 가능한 태스크들을 관리한다.

Linux CFS는 이 태스크들의 virtual runtime을 비슷하게 유지하면서 per-core 공정할당 스케줄링을 수행한다. 태스크의 virtual runtime은 다음의 식으로 정의된다.

$$v_i(t) = \frac{w_0}{w(\tau_i)} \times c_i(t) \quad (4)$$

$c_i(t)$ 는 시간 t동안 태스크 τ_i 가 받은 CPU 시간 을 나타낸다. 위 식에서 나타낸 것처럼, 태스크의 virtual runtime은 해당 태스크의 가중치에 반비례하며, CPU 시간에 비례한다.

제 2 절 공정할당의 정의

비대칭 멀티코어 시스템의 공정성(fairness)의 개념을 유도하기 위하여, 우선 현재 사용되는 대칭 멀티코어 시스템에서의 공정성의 정의를 살펴본다 [1][8].

태스크 집합 T를 실행시키고 있는 대칭 멀티코어 시스템을 생각한다. $c_i(t)$ 는 시간 t 동안 태스크 $\tau_i \in T$ 가 받은 CPU 시간을 나타낸다. 대칭 멀티코어 시스템에서 완벽한 공정할당 스케줄러는 다음과 같이 정의한다 [1][8].

정의 1. 구간 $[0, t]$에서 지속적으로 수행 가능한 태스크 $\tau_i \in T$와 $\tau_j \in T$가 존재할 때, 대칭 멀티코어 시스템에서의 완벽한 공정할당 스케줄러는 다음을 만족한다.

$$\frac{c_i(t)}{c_j(t)} = \frac{w(\tau_i)}{w(\tau_j)} \tag{5}$$

이어, 스케일된 CPU 시간을 정의 한다. 태스크 집합 T를 실행시키고 있는 비대칭 멀티코어 시스템을 고려한다. $\hat{c}_i(t)$ 는 시간 t 동안 태스크 τ_i 가 받은 CPU 시간을 나타낸다. 스케일된 CPU 시간은 아래와 같이 정의된다.

정의 2. 비대칭 멀티코어 시스템에서 태스크 τ_i 가 구간 $[0, t]$에서 받은 스케일된 CPU 시간은 다음과 같이 나타낸다.

$$\hat{c}_i(t) = \int_0^t R_i(t)dt \tag{6}$$

이 정의를 기반으로, 우리는 비대칭 멀티코어 시스템에서의 완벽한 공정할당 스케줄러를 다음과 같이 정의한다.

정의 3. 구간 $[0, t]$에서 지속적으로 수행 가능한 태스크 τ_i와 τ_j가 존재할 때, 비대칭 멀티코어 시스템에서의 완벽한 공정할당 스케줄러는 다음을 만족한다.

$$\frac{\hat{c}_i(t)}{\hat{c}_j(t)} = \frac{w(\tau_i)}{w(\tau_j)} \tag{7}$$

본 학위논문에서 제안하는 접근법은 Linux CFS의 per-core 공정할당 스케줄링에 기반을 두고 있고, 이는 virtual runtime을 사용한다. 따라서 우리는 근본적인 CFS의 virtual runtime의 개념을 수정하여 성능 비대칭 특성을 스케줄링에 반영한다. 이어서, 스케일된 virtual runtime을 아래와 같이 정의한다.

정의 4. 시간 t 에서 태스크 τ_i 의 스케일된 virtual runtime SVR(Scaled Virtual Runtime)은 다음과 같다.

$$\hat{v}_i(t) = \frac{1}{w(\tau_i)} \times \hat{c}_i(t) \tag{8}$$

만일 비대칭 멀티코어 시스템에서 사용되는 SVR 기반 스케줄러가 모

든 태스크의 SVR을 같게 한다면 위 식(7)을 만족하기에 완벽한 공정할당 스케줄러라고 할 수 있다.

제 3 절 문제 정의

비대칭 멀티코어 시스템의 모든 태스크들이 임의의 시간 시간 t 에서 동일한 SVR 값을 갖는다면, 이 시스템은 완벽한 공정할당 특성을 나타낸다고 말 할 수 있다. 하지만 실제 상황에서 이러한 스케줄러를 구현한다는 것은 비 현실적이다. 따라서, 본 학위논문에서는 이러한 스케줄러에 기능적으로 근접한 스케줄러를 제안한다. 구체적으로, 태스크간 SVR 값 차이가 상수 값 이내로 제한되게 함으로써 이를 구현한다.

이러한 스케줄러를 구현함에 있어서, 코어들 사이에서 태스크들이 이주되는 것은 필연적으로 발생한다. 그리고 앞서 설명한대로, 빅리틀 아키텍처에서는 두 가지 이주정책이 있다: (1) 클러스터간 태스크 이주정책과 (2) 클러스터 내부 부하분산정책. GTS 모드를 위한 Linaro의 스케줄링 프레임웍은 클러스터간 태스크 이주정책을 사용한다. 이는 태스크의 현재 상태를 파악한 후, 알맞은 코어 타입을 태스크에 할당하여 성능을 최대한 끌어올리기 위함이다.

본 학위논문에서는 성능을 떨어뜨리지 않는 범위에서 태스크간 SVR 차이를 상수 값 이내로 제한함을 목적으로 한다. 이를 위하여, 클러스터간 태스크 이주정책은 원래의 정책을 그대로 사용하고, 클러스터 내부 부하분산 정책을 개선하여, 각 클러스터 내부의 태스크간 SVR 차이를 상수 값 이내로 제한한다.

어떤 클러스터에 존재하는 모든 태스크가 태스크 집합 T 에 속하고, 그들 중 임의의 두 태스크 τ_i 와 τ_j가 선택되었다고 하자. 이때 두 태스

크의 SVR 차이를 $\left|\hat{v}_{i,j}(t)\right|$로 정의한다. 이어서, $\hat{v}_{max}(t)$를 다음과 같이 정의한다:

$$\hat{v}_{max}(t) = \max_{\{\tau_i,\tau_j\}\in T}\left(\left|\hat{v}_{i,j}(t)\right|\right) \leq C \quad (9)$$

위 식을 이용하여 구현하고자 하는 스케줄러의 목적을 설명하면, 각 클러스터의 $\hat{v}_{max}(t)$가 상수 C보다 작은 값으로 제한되게 하는 것이다.

제 4 절 해결책

본 장에서는 앞 절에서 정의된 문제를 해결하는 해결책을 제시한다. 본 학위논문에서 제안하는 기법은 크게 두 가지로 구성되어 있다: (1) relative performance를 사용하여 각 태스크의 SVR을 산정한다. (2) 클러스터 내부에 존재하는 실행 가능한 태스크들의 SVR 값을 기반으로, 코어간 태스크들을 이주 시킨다. 이를 통하여 클러스터 내부의 태스크간 SVR 값 차이를 상수 값 이내로 제한한다.

제안된 기법이 기존의 Linaro 스케줄링 프레임웍과 부작용 없이 동작하기 위하여, SVR calculator 라는 새로운 모듈을 각 코어 별로 존재하는 per-core 스케줄러에 추가하였다. 또한, 각 클러스터에 존재하는 가중치 기반 부하분산 정책을 SVR 기반 부하분산 정책으로 교체하였다.

동적으로 변화하는 태스크의 phase 특성을 반영하기 위하여, SVR calculator는 매 스케줄링 tick마다 현재 수행되고 있는 태스크의 relative performance를 측정한다. 본 연구에서 스케일된 CPU 시간과 누적된 형태의 SVR 값을 구하기 위하여, 몇 개의 kernel 함수를 수정하

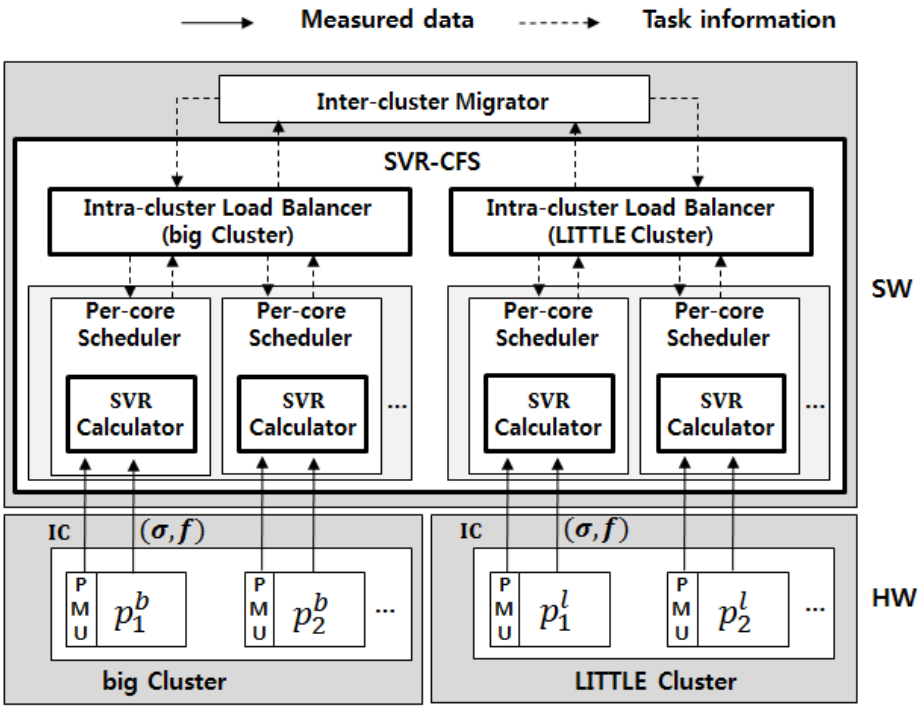


그림 13. 공정할당 스케줄링을 위한 해결책 개괄 도표

였다. SVR 기반 부하분산 기법은 각 클러스터 내부에 존재하며, 같은 클러스터 내부에서 실행 가능한 태스크들의 SVR 값 차이를 상수 값 이내로 제한한다. 본 연구에서는 이렇게 도출된 스케줄러를 SVR-CFS라고 정의한다.

그림 13은 제안하는 기법의 전체적인 아키텍처를 보여준다. 그림에서 알 수 있듯이, 제안된 기법은 클러스터 구조의 빅리틀 멀티코어 시스템 위에서 구현되었다. SVR calculator는 하드웨어로부터 수행된 IC(instruction counts) 값, 코어 타입(σ), 동작 주파수(f)등의 정보를 받는다. 이어서, 현재 수행중인 태스크의 relative performance를 계산한다. 이렇게 계산된 결과를 사용하여, 그 태스크의 SVR 값으로 확장한

다.

이 값을 기반으로, SVR-CFS 각 코어의 실행 큐에 속해있는 실행 가능한 태스크들에 대하여 per-core 스케줄링을 수행한다. 미리 설정된 주기 값을 기반으로 매 주기마다, SVR 기반 부하분산 정책은 클러스터 내부의 태스크들에 대하여 SVR 값 차이가 상수 값 이내로 제한되게 부하를 분산시킨다. 결과적으로 본 논문에서 제안하는 기법은 공정할당 스케줄링을 수행한다.

이어지는 하기 절의 4.1 은 SVR을 구하는 방법을 기술한다. 4.2는 SVR 기반 per-core 스케줄링을 설명하고, 4.3은 SVR 기반 부하분산 정책에 사용된 알고리즘들을 자세히 설명한다. 마지막으로 4.4는 제안된 알고리즘을 수학적으로 분석하고 이를 검증한다.

4.1 SVR 계산

태스크의 SVR을 구하기 위해서는 먼저 그 태스크가 나타내는 relative performance를 구해야 한다. 태스크 τ_i의 $R_i(t)$는 시간 t에서 현재 코어가 나타내는 성능과, 주파수가 $f_{min}^{l} \in F(P_l)$ 상태에서 측정된 성능의 비율을 나타낸다. 앞서 설명한 IPT를 이용하여 이를 정의하면 다음과 같다.

$$R_i(t) = \frac{IPT_{real}(\tau_i, t)}{IPT_{base}(\tau_i, t)} \tag{10}$$

위 식에서, $IPT_{real}(\tau_i, t)$는 시간 t 에서 측정한 실제 IPT를 뜻하고, $IPT_{base}(\tau_i, t)$는 시간 t에서 코어가 $f_{min}^{l} \in F(P_l)$로 동작했음을 가정했을

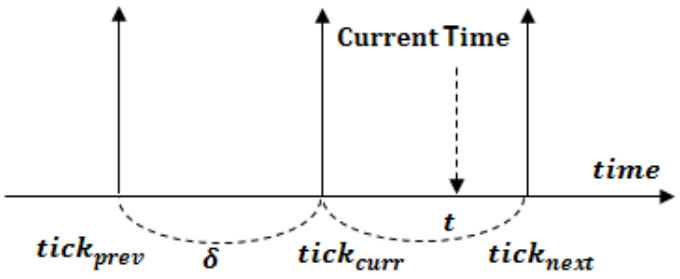


그림 14. 현재시간 t 와 스케줄링 tick의 관계

때의 IPT이다. 이 두 값들 모두 가장 최근 두 개의 스케줄링 tick 사이에서 측정된 IPT를 나타낸다. 현재 시간 t와 스케줄링 tick의 관계를 보다 쉽게 이해하기 위해서, 이를 그림 14에 나타내었다. 가장 최근 두 개의 스케줄링 tick이란 그림에서 볼 수 있는 것처럼 $tick_{prev}$와 $tick_{curr}$를 뜻한다.

그림에서 알 수 있듯이, $R_i(t)$는 매 스케줄링 tick마다 갱신된다. 태스크 τ_i가 최근 두 개의 스케줄링 tick사이에 동작하지 않았다면, $R_i(t) = 0$을 나타낸다. 그 이유는 어떠한 instruction도 코어에서 수행되지 않았기 때문이다.

$IPT_{real}(\tau_i, t)$ 는 그림 13에서 알 수 있듯이, 각 코어 별 PMU(performance monitoring unit)를 통하여 구할 수 있다. PMU는 코어가 수행한 instruction, cache miss등의 정보를 레지스터에 카운터 값으로 저장한 후 알려주는 하드웨어 장치이다 [5][6].

하지만, $IPT_{base}(\tau_i, t)$는 $IPT_{real}(\tau_i, t)$의 경우처럼 바로 구할 수 없다. 그 이유는 태스크 τ_i가 시간 t에서 $f_{min}^{l} \in F(P_l)$로 동작하지 않았을 확률이 높기 때문이다. 이 값을 구하기 위해서 본 연구에서는 간단한 linear

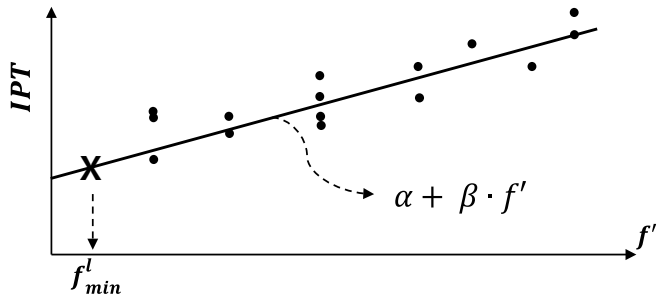


그림 15. IPT 와 동작 주파수 f'

regression 방법을 사용하였다. 사용된 방법은 OLS(Ordinary Least-Squares) regression이다 [43].

$F(P_b)$에 속한 주파수와 $F(P_l)$에 속한 주파수 중 몇 개의 값은 서로 같은 값을 갖는다. 하지만 코어의 성능 차이로 인하여 어떤 태스크의 relative performance는 주파수가 같더라도 다르다. 따라서, 빅코어와 리틀코어의 주파수 값들이 서로 겹치지 않고 하나의 도메인에서 표현되도록 아래와 같은 식으로 나타낸다.

$$f' = \varepsilon \cdot f$$

위 식에서 f는 코어의 동작주파수를 나타내고, ε는 미리 정해진 상수를 나타낸다. 본 연구에서 ε는 동작주파수가 $f \in F(P_l)$일 경우 1의 값을 갖고 $f \in F(P_b)$일 경우 1.8을 갖게 하였다. 이 값들은 그림 16을 기준으로 산정한 값이며 [44], 그림에 대한 자세한 설명은 이 세부 절의 마지막 부분에 기술하였다.

이렇게 얻어진 f' 와 태스크의 IPT 데이터를 이용하여 OLS

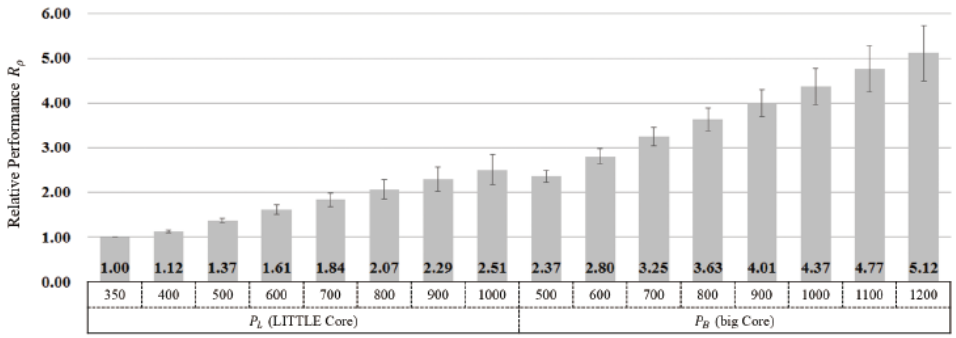


그림 16. 동작 주파수 f'에 따른 R_ρ

regression를 수행하는 방법은 다음과 같다. 첫째, f'에 따른 어떤 태스크의 IPT 데이터를 최근 k 스케줄링 tick동안 저장한다. 이때, k값은 본 연구에서 20으로 설정하였다. 이어, 다음의 식이 나타내는 OLS regression의 α, β 값들을 구한다 [43].

$$IPT = \alpha + \beta \cdot f' \tag{11}$$

마지막으로 위 식의 f'에 f_{min}^{l}값을 대입함으로써 $IPT_{base}(\tau_i, t)$값을 구한다. 이를 그림 15에 나타내었다.

시스템이 $R_i(t)$값을 구하기 위한 충분한 양의 IPT와 f' 데이터가 없을 경우가 발생할 수 있다. 즉, 해당 태스크가 시작한지 얼마 되지 않았거나, 혹은 한 개의 고정된 주파수로 동작했을 때이다. 이러한 경우, SVR calculator는 시스템이 수행되기 전에 미리 얻어진 R_ρ 값을 사용한다.

그림 16는 R_ρ와 f'의 관계를 나타낸다. 이를 위하여 본 연구에서는 SPEC CPU2006 벤치마크 [45][46] 중 bzip2, xalanc, h.264ref, sjeng을 사용하고, 이들의 각 f'에 따른 완료시간 평균을 구하였다. $f' = f_{min}^{l}$에

서 4가지 벤치마크의 평균 완료시간을 $R_\rho = 1$ 로 정의하고, 나머지 주파수 f' 에서 측정된 평균 완료시간들은, 이 값을 기준으로 normalize한 값이다 [44].

4.2 SVR 기반 per-core 스케줄링

앞서 설명한 태스크들의 SVR 값들에 기반하여, SVR-CFS는 per-core 공정할당 스케줄링을 수행한다. 태스크들은 그들에게 부여된 time slice를 다 소진했을 경우, 현재 수행된 코어의 실해 큐로 돌아가게 된다. 이때, 실행 큐 내부에서의 위치는 태스크의 SVR 값에 의하여 결정된다.

SVR-CFS역시 원래의 CFS와 마찬가지로 SVR 크기 순서로 정렬된 red-black 트리를 사용한다. 따라서 실해 큐로 복귀하는데 걸리는 시간은 $O(log\, n)$ 이며, n은 트리에 속해있는 태스크의 수이다. SVR-CFS가 다음 수행시킬 태스크를 찾을 때는 실행 큐의 가장 작은 SVR 값을 갖는 트리의 노드를 찾으면 된다. SVR-CFS는 트리의 가장 아래쪽에 위치한 노드들 중 가장 왼쪽에, SVR값이 가장 작은 태스크 노드를 할당한다. 따라서 실행 큐로 복귀할 때와는 대조적으로 이때의 수행시간은 $O(1)$이다.

본래의 CFS에서는, 태스크의 virtual runtime(virtual runtime)을 관리할 때, 누적된 형태로 관리하지 않는다. CFS는 부하분산을 위하여 어떤 태스크가 실행 큐로 진입하거나 빠져나갈 때, 원래의 virtual runtime을 갱신한다. 그 이유는, 실행 큐내부에서 태스크들이 virtual runtime을 취할 때 상대적인 값을 갖게 하기 위해서이다 [23][24].

시간 t_1에서 태스크 τ_i가 실행 큐 q_j에서 빠져나올 때, 이 태스크의

virtual runtime 은 본래의 CFS에서 아래 식처럼 계산된다.

$$v_i^{'}(t_1) = v_i(t_1) - v_{min}^{j}(t_1) \quad (12)$$

위 식에서, $v_{min}^{j}(t_1)$은 시간 t_1에서 태스크 τ_i가 실행 큐 q_j에 존재할 때, q_j에 속해있는 태스크들 중 가장 작은 virtual runtime을 갖는 태스크의 virtual runtime이다.

반대로, 시간 t_2에서 태스크 τ_i가 실행 큐 q_j에 삽입될 때, 이 태스크의 virtual runtime은 아래 식처럼 본래의 값 대신 갱신된다.

$$v_i(t_2) = v_i^{'}(t_2) + v_{min}^{j}(t_2) \quad (13)$$

위 식들을 쉽게 이해하기 위해서, 그림 17에 CFS의 virtual runtime 관리 기법을 나타내었다. 왼쪽의 그림은 시간 t_1에서 'Task 1'이 실행 큐 q_0에서 빠져나올 때, $v_{min}^{0}(t_1)$ 값이 1000에서 0으로 바뀌는 과정을 설명하고 있다. 이렇게 바뀐 이유는, 위 식 (12)에 나타내었듯이, q_0의 최소 virtual runtime이 'Task 1' 자신 본래의 virtual runtime이기 때문이다.

오른쪽 그림은 위와 반대로, virtual runtime 값이 50인 'Task 3'이 q_1에 삽입될 때의 동작을 나타낸다. 위 식 (13)에 나타내었듯이, q_1의 최소 virtual runtime이 900이기 때문에, 50에서 950으로 갱신되어 삽입된다.

이처럼 본래의 CFS는 virtual runtime을 계산할 때, 어떤 태스크의 모든 과거 수행시간 이력을 보존하지 않는다. 따라서, 이러한 방법은 본 학위논문에서 제안하는 기법에 적용할 수 없다.

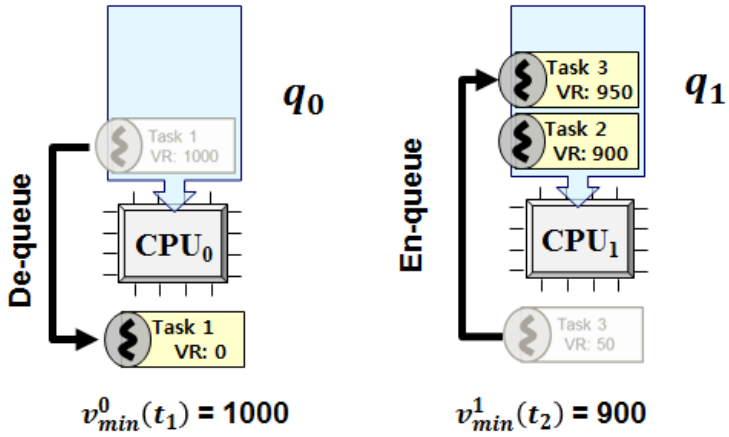


그림 17. CFS의 virtual runtime 관리 기법

정의 2 에 기술된 것 처럼, 태스크 τ_i의 스케일된 CPU 시간을 구하기 위해서는 이 태스크의 모든 수행시간 이력을 보존해야한다. 식 (6)에 나타낸 것처럼, $\hat{c}_i(t)$는 태스크 τ_i의 $[0,t]$ 동안의 수행 이력을 가지고 있다. 이는 태스크가 생성된 후부터 시간 t까지 모든 부분 구간 수행 이력을 가지고 있다. 또한, 이는 각 부분 구간 동안, 그 태스크에 할당된 동작 주파수, 코어타입, 동작특성 등을 모두 가지고 있다. 따라서, 본 연구에서는 CFS의 virtual runtime 갱신 방식을 수정하여, SVR-CFS는 시스템의 모든 태스크들의 SVR을 누적된 형태로 관리하도록 하였다.

4.3 SVR 기반 부하분산 알고리즘

본 절에서는 같은 클러스터 내부의 어떤 태스크 쌍도 그 SVR 차이가 상수 값 이내로 제한되는 SVR 기반 부하분산 알고리즘에 대하여 기술한

다. 제안된 알고리즘은 빅클러스터 및 리틀클러스터의 구분 없이 적용되는 알고리즘이다. 따라서, 알고리즘 설명에 필요한, 표 3에 정의된 각 클러스터별 수학적 기호를 단순화 시킬 필요가 있다. 이를 클러스터 구분 없이 단순화하여 표현하면 다음과 같다.

m개의 동일한 코어로 이루어진 임의의 클러스터는 n개의 실행 가능한 태스크로 이루어진 태스크 집합 $T = \{\tau_1, \tau_2, \dots, \tau_n\}$를 수행하고 있다. 이때, 이 클러스터의 실행 큐 집합과 코어별 실행 큐는 $Q = \{q_1, q_2, \dots, q_m\}$로 나타낸다. 또한, 각 실행 큐 $q_k \in Q$는 태스크 그룹 G_k를 가지고 있다.

자세한 알고리즘의 설명에 앞서, feasible set을 정의한다. 이는 멀티코어 아키텍처의 공정할당 스케줄러가 스케줄링이 가능한 태스크 집합을 뜻한다. 본 연구에서 feasible set의 정의는 다음과 같다. 한 태스크 집합 $T = \{\tau_1, \tau_2, \dots, \tau_n\}$가 m개의 코어로 이루어진 클러스터 내부에서 운용될 때, 다음 조건을 만족하면 feasible set이라고 부를 수 있다 [23].

$$\forall i: 1 \le i \le n, \qquad w(\tau_i) \le \frac{\sum_{j=1}^{n} W(\tau_j)}{m} \tag{14}$$

위 식의 의미는 다음과 같이 설명할 수 있다. 어떤 클러스터 내부의 어떤 태스크 τ_i의 가중치는 클러스터내부에 존재하는 모든 태스크들의 가중치 합의 $1/m$ 크기보다 작거나 같아야 한다는 뜻이다. 본 연구에서 제안하는 기법은 이를 만족하는 태스크 집합 $T = \{\tau_1, \tau_2, \dots, \tau_n\}$에 대해서 동작한다.

본 논문의 이후에 기술되는 내용의 이해를 돕고자 어떤 태스크의 스케일된 가중치와 어떤 태스크 그룹의 스케일된 부하를 정의한다.

정의 5. 어떤 태스크 τ_i의 r번째 부하분산 주기$[(r-1)\lambda,\ r\lambda]$ 동안의

스케일된 가중치는 다음과 같다.

$$\widehat{w}(\tau_i, r) = \frac{w(\tau_i)}{R_i^{avg}(r)}$$

위 식에서 $R_i^{avg}(r)$는 τ_i 가 $[(r-1)\lambda,\ r\lambda]$ 동안 동작할 때, 평균 relative performance $R_i(t)$ 값을 나타낸다.

정의 6. 어떤 태스크 그룹 G_k의 r번째 부하분산 주기$[(r-1)\lambda,\ r\lambda]$ 동안의 스케일된 부하는 다음과 같다.

$$\hat{L}_k(r) = \sum_{\tau_i \epsilon G_k} \widehat{w}(\tau_i, r)$$

대칭 멀티코어 아키텍처에서는 각 태스크에게 주어지는 CPU시간은 단순하게 가중치에 비례한다. 이와 달리, 비대칭 멀티코어 아키텍처에서는 CPU 시간은 코어의 컴퓨팅 능력이 반영된 가중치에 비례해야 한다. 이를 위하여 본 연구에서 태스크의 스케일된 가중치와 태스크 그룹의 스케일된 부하를 **정의 5** 와 **정의 6**에 나타내었다.

식 (5),(6),(7)을 사용하여 $c_i(t)$를 $w(\tau_i)$와 $R_i(t)$로 나타내면, 보다 직관적으로 알 수 있다. 이로써 비대칭 멀티코어 아키텍처의 완벽한 공정 할당 스케줄링을 위해서는 어떤 태스크 τ_i의 CPU 시간 $c_i(t)$는 매 부하분산 주기 동안 스케일된 가중치에 비례해야 함을 알 수 있다. 본 학위논문의 이후에서는, 수학적 용어 표현을 간단히 하기 위해서 $\hat{L}_k(r)$를 $\hat{L}_k$로 간략히 표기한다.

ALGORITHM 2. SVR-BASED LOAD BALANCING

Input: A set of tasks in a cluster: T
The number of cores: m

1: H = **SORT**(T)
2: $G \leftarrow$ **SPLIT**(H, m)
3: $G \leftarrow$ **ADJUST**(G)
4: **return** G

ALGORITHM 2는 top-level SVR-BASED LOAD BALANCING 알고리즘을 나타내고 있다. 이 알고리즘은 매 부하분산 주기 λ마다 불려지고, 그때마다 각 태스크들의 실행 큐를 결정한다. 알고리즘의 입력으로 두 가지 인자를 받게 된다: (1) 어떤 클러스터 내부의 실행 가능한 태스크 집합 T와 (2) 클러스터가 가지고 있는 코어 수 m. 이 알고리즘은 m개의 태스크 그룹 $G = \{G_1, G_2, \ldots, G_m\}$를 출력하고, 다음의 성질을 만족한다.

(1) $1 \le k < m$를 만족하는 모든 k에 대하여, 태스크 그룹 G_k에 속한 모든 태스크의 SVR 값은 G_{k+1}에 속한 어떠한 태스크의 SVR 값보다 작거나 같다.

(2) $\hat{L}_1 \le \hat{L}_2 \le \cdots \le \hat{L}_m$.

(3) $0 \le \hat{L}_{k+1} - \hat{L}_k \le 2w_{max}$.

최종 출력된 모든 G_k에 대하여, 속한 태스크들은 실행 큐 q_k에 할당된다. 그리고 이러한 $G = \{G_1, G_2, \ldots, G_m\}$이 존재하는 경우, 태스크 집합 T를 balanceable set이라 부른다.

이 알고리즘은 먼저 라인 1에 나타내 것처럼 merge-sort를 수행한다. 이때, SVR 값들의 오름차순으로 각 태스크들을 정렬한 후 시퀀스 H를 생성한다. 그 후, SPLIT을 수행한다 (라인 2). 이는 정렬된 시퀀스 H를 m 태스크 그룹으로 나눈다. 이때 생성된 태스크 그룹의 집합은 $G = \{G_1, G_2, \ldots, G_m\}$이며, 각 인접한 태스크 그룹간의 스케일된 부하 차이는 상수 값 이내로 제한된다. 라인 3에 기술된 ADJUST는 라인 2에 기술된 SPLIT 의 결과인 $G = \{G_1, G_2, \ldots, G_m\}$를 입력으로 받는다. 그리고 이를 조정하여, 위에 기술된 SVR-BASED LOAD BALANCING의 세 가지 성질을 만족하는 $G = \{G_1, G_2, \ldots, G_m\}$를 최종 출력하게 된다.

앞서 기술된 SVR-BASED LOAD BALANCING의 성질 중 (1)과 (2)는 다음을 만족한다. 큰 SVR 값을 가진 태스크들은 더 큰 스케일된 부하 값을 가지는 코어에서 수행되므로, 다음 부하분산 주기 시점까지 더 느리게 SVR 값이 증가하게 된다. 또한, (3)은 큰 SVR 값을 가진 태스크들이 작은 SVR 값을 가진 태스크들 보다 너무 느리게 SVR 값이 증가하지 않게 제한을 둔다.

ALGORITHM 3은 SPLIT의 의사 코드를 나타내고 있다. 이는 정렬된 시퀀스 H를 입력으로 받은 후 m개의 태스크 그룹을 생성한다. 이렇게 나뉘어진 m개의 태스크 그룹은 $G_1, G_2, \ldots, G_m$형태로 정렬이 되고, 각 인접한 태스크 그룹의 스케일된 부하의 차이는 상수 값 이내로 제한된다.

ALGORITHM 3. SPLIT

Input: A sequence of tasks sorted in ascending order with SVR: $H =< \tau_1, \tau_2, \ldots, \tau_n >$, The number of cores: m

SPLIT(H, m)

1: $G_1 \leftarrow \emptyset,\ G_2 \leftarrow \emptyset, \ldots,\ G_m \leftarrow \emptyset$
2: $i \leftarrow 1$
3: $\hat{L}_{acc} \leftarrow 0$
4: **for** $k \leftarrow 1$ **to** $m-1$ **do**
5: $\hat{L}_k^E \leftarrow$ **CALCEXPECTEDLOAD**(H, m, $\hat{L}_{acc}$, k)
6: **while** $\hat{L}_k + W(\tau_i) \leq \hat{L}_k^E$ **do**
7: $G_k \leftarrow G_k \cup \{\tau_i\}$
8: $i \leftarrow i+1$
9: **end while**
10: $\hat{L}_{acc} \leftarrow \hat{L}_{acc} + \hat{L}_k$
11: **end for**
12: **return** $\{G_1, G_2, \ldots, G_m\}$

CALCEXPECTEDLOAD(H, m, $\hat{L}_{acc}$, k)

13: $\hat{L} \leftarrow$ sum of scaled weights of tasks in H
14: $\hat{L} \leftarrow (\hat{L} - \hat{L}_{acc})/(m-k+1)$
15: **return** $\hat{L}$

이 알고리즘은 각 태스크 그룹 $G_1, G_2, \ldots, G_m$에 반복적으로 태스크를 할당한다. 이 동작을 수행하기 위해서, 알고리즘은 먼저 라인 5처럼 CALCEXPECTEDLOAD 함수를 호출한다. 이 함수는 태스크 그룹 G_k가 취할 수 있는 스케일된 부하의 기대값을 구하며, 이 기대값은 다음과 같이 정의된다.

$$\hat{L}_k^E = \frac{\sum_{j=k}^{m} \hat{L}_j}{m-k+1} \tag{15}$$

위 식에서 알 수 있듯이, $\hat{L}_k^E$ 는 $m-k+1$ 개의 태스크 그룹인 $G_k, G_{k+1}, \ldots, G_m$에서 각 태스크 그룹에 할당 가능한 스케일된 부하의 평균 값을 나타낸다.

이렇게 구해진 값을 사용하여 $\hat{L}_k$ 가 $\hat{L}_k^E$ 보다 작거나 같을 경우, $G_1, G_2, \ldots, G_{k-1}$에 속하지 않는 태스크 중 가장 작은 SVR 값을 갖는 태스크를 반복해서 G_k에 할당한다 (lines 6~9).

SPLIT 은 $G = \{G_1, G_2, \ldots, G_m\}$에 속한 모든 태스크 그룹에게 비슷한 스케일된 로드가 부여되도록 노력한다. 이러한 이유로 각 인접한 태스크 그룹들간의 스케일된 부하 차이는 상수 값 이내로 제한될 수 있다. 이에 관한 자세한 설명과 증명은 다음 세부 절에서 하기로 한다.

ALGORITHM 4 는 ADJUST 의 의사 코드를 나타낸다. 이 알고리즘의 목적은 $G_1, G_2, \ldots, G_m$ 태스크 그룹들이 갖는 스케일된 부하 값들이 $\hat{L}_1 \le \hat{L}_2 \le \cdots \le \hat{L}_m$ 의 관계를 만족하도록 하는 것이다. SPLIT 의 출력이 ADJUST 의 입력으로 부가되며, 이 또한 m 개의 태스크 그룹 $G = \{G_1, G_2, \ldots, G_m\}$를 출력한다.

이 알고리즘은 반복적으로 CHECKANDMOVETASKS 함수를 호출하며, 이때 입력을 G_{m-1} 에서 G_1 순으로 변화 시킨다 (lines 2~4). 이 함수의 주된 동작은 k 값을 입력으로 받아서 태스크 그룹 $G_k, G_{k+1}, \ldots, G_m$ 의 스케일된 부하의 순서를 $\hat{L}_k \le \hat{L}_{k+1} \le \cdots \le \hat{L}_m$ 로 하게 한다. 따라서 ADJUST 의 모든 동작이 완료된 시점에는 $\hat{L}_1 \le \hat{L}_2 \le \cdots \le \hat{L}_m$ 의 성질을 얻을 수 있다.

CHECKANDMOVETASKS 이 불릴 때 마다, $\hat{L}_k$ 와 $\hat{L}_{k+1}$ 의 대소 관계를 살핀다. 이때 $\hat{L}_k \leq \hat{L}_{k+1}$ 이 만족한다면, 이 함수는 아무런 동작을 수행하지 않고 빠져 나온다. 하지만 $\hat{L}_k > \hat{L}_{k+1}$ 경우에는, G_k 에 속한 태스크 중 가장 큰 SVR 값을 가진 태스크가 G_{k+1} 태스크 그룹으로 $\hat{L}_k \leq \hat{L}_{k+1}$를 만족할 때까지 옮겨지게 된다 (lines 6~13).

이어서, 이 함수는 G_k 와 G_{k+1} 의 사이에서 발생한 태스크 이동이 G_{k-1} 와 G_{k+2} 에 끼친 영향을 조사한다. 이를 위하여 CHECKANDMOVETASKS 함수의 입력으로 $k-1$ 와 $k+1$ 을 부가하여 CHECKANDMOVETASKS 함수를 재귀적으로 호출한다 (lines 14~19). 이렇게 함으로써, k 값을 입력으로 받은 CHECKANDMOVETASKS 함수는 $\hat{L}_{k-1} \leq \hat{L}_k \leq \hat{L}_{k+1} \leq \hat{L}_{k+2}$ 의 성질을 만족 시킨다.

위에서 설명한 ADJUST 가 모든 동작을 완료한 시점에는, 앞서 기술한 SVR-BASED LOAD BALANCING 의 세가지 성질을 만족하게 된다. 이에 대한 자세한 수학적 분석 및 검증은 다음 세부 절에서 설명한다.

ALGORITHM 4. ADJUST

Input: Set of task groups: $G = \{G_1, G_2, \ldots, G_m\}$

ADJUST(G)

1: **for** $k \leftarrow m-1$ **down to** 1 **do**
2: **CHECKANDMOVETASKS**(G, k)
3: **end for**
4: **return** G

CHECKANDMOVETASKS (G, k)

5: **if** $\hat{L}_k > \hat{L}_{k+1}$ **then**
6: **while** $\hat{L}_k > \hat{L}_{k+1}$ **do**
7: **if** $k = 1$ **and** there is only one task in G_1 **then**
8: **print** *"Given task set is not a balanceable set"*
9: **return** $\emptyset$
10: **else**
11: move the task with the largest SVR in G_k to G_{k+1}
12: **end if**
13: **end while**
14: **if** $k-1 \geq 1$ **then**
15: **CheckAndMoveTasks**$(G, k-1)$
16: **end if**
17: **if** $k+1 \leq m-1$ **then**
18: **CheckAndMoveTasks**$(G, k+1)$
19: **end if**
20: **end if**

4.4 알고리즘의 수학적 분석 및 검증

본 절에서는 본 학위논문에서 제안된 SVR-BASED LOAD BALANCING 알고리즘을 자세히 수학적으로 분석한다. 분석함에 있어서 구체적으로, 같은 클러스터 내부에서 임의의 두 태스크는 임의의 시간 t 에서 $\hat{v}_{max}(t)$ 값이 상수 값 이내로 제한됨을 보인다.

이를 위하여 본 절에서는 첫째, 앞서 설명한 SVR-BASED LOAD BALANCING 알고리즘의 세 가지 성질을 증명한다. SPLIT과 ADJUST 를 수행하면서, 클러스터 내부의 모든 태스크들의 SVR 값에 의한 순서는 변하지 않는다. 따라서 (1)은 당연히 만족된다. ADJUST가 가지고 있는 본연의 특징에 의하여 (2) 역시 만족된다. 따라서 본 절에서는 성질 (3)이 만족됨을 보임으로써 증명을 완료한다.

LEMMA 4.1. 동일 클러스터 내부에 존재하는 임의의 인접하고 있는 태스크 그룹의 스케일된 부하의 차이는 $2w_{max}$보다 작거나 같고, 이는 다음의 식으로 표현된다.

$$0 \leq \hat{L}_{k+1} - \hat{L}_k \leq 2w_{max} \text{ for } 1 \leq k < m$$

PROOF. 위 식에 대한 증명은 두 단계의 스텝으로 이루어진다: (a) SPLIT에 의하여 인접한 태스크 그룹간의 스케일된 부하의 차이는 상수 값 이내로 제한된다. 그리고 (b) LEMMA 4.1은 ADJUST에 의하여 만족한다.

Step (a): SPLIT 은 반복적으로 태스크 그룹 G_k에 가장 작은 SVR 값을 갖는 태스크를 삽입 시킨다. 이를 수행함에 있어서, $\hat{L}_k$가 $\hat{L}_k^E$를 넘지 않을

때까지 수행한다. 따라서 다음의 식을 얻을 수 있다.

$$\hat{L}_k^E - w_{max} \le \hat{L}_k \le \hat{L}_k^E \tag{16}$$

식 (16)에서 k를 $k+1$로 대체한 후, 식 (16)에서 이를 빼면 다음의 관계를 얻을 수 있다.

$$\hat{L}_{k+1}^E - \hat{L}_k^E - w_{max} \le \hat{L}_{k+1} - \hat{L}_k \le \hat{L}_{k+1}^E - \hat{L}_k^E + w_{max} \tag{17}$$

다음으로, 식 (17)에 나타낸 $\hat{L}_{k+1} - \hat{L}_k$의 범위를 구하기 위해서 $\hat{L}_{k+1}^E - \hat{L}_k^E$의 범위를 구한다. 식 (15)에 정의된 $\hat{L}_k^E$에서, k를 $k+1$로 대체하면 다음 식을 얻을 수 있다.

$$L_{k+1}^E = \frac{\sum_{j=k+1}^{m} \hat{L}_j}{m-k} = \frac{(m-k+1)\hat{L}_k^E - \hat{L}_k}{m-k}$$

$$\Rightarrow \hat{L}_{k+1}^E - \hat{L}_k^E = \frac{\hat{L}_k^E - \hat{L}_k}{m-k} \tag{18}$$

식 (18)을 사용하여 식 (17)을 다시 쓰면 다음과 같다.

$$\frac{\hat{L}_k^E - \hat{L}_k}{m-k} - w_{max} \le \hat{L}_{k+1} - \hat{L}_k \le \frac{\hat{L}_k^E - \hat{L}_k}{m-k} + w_{max}$$

식 (15)와 SPLIT 의 정의된 동작에 의하여, $0 \le \hat{L}_k^E - \hat{L}_k \le w_{max}$ 은 당연히 만족한다. 결과적으로, 이를 위 식에 대입하면 다음의 식을 얻을 수 있다.

$$-w_{max} \le \hat{L}_{k+1} - \hat{L}_k \le \frac{w_{max}}{m-k} + w_{max} \le 2w_{max}$$

Step (b): ADJUST 는 $\hat{L}_{k+1} - \hat{L}_k < 0$인 경우, 반복적으로 G_k에서 G_{k+1}로 태스크를 이동 시킨다. 따라서 $\hat{L}_{k+1} - \hat{L}_k$가 취할 수 있는 하한 값은 0 이 된다. 또한, 태스크 한 개가 이동될 때 $\hat{L}_{k+1} - \hat{L}_k$가 취할 수 있는 최대 값은 $2w_{max}$ 이다. 따라서 $\hat{L}_{k+1} - \hat{L}_k$ 가 취할 수 있는 값의 상한 값은 SPLIT 의 상한 값과 동일하다. 결론적으로, 이 lemma 는 참이다. □

LEMMA 4.2. 동일 클러스터 내부의 임의의 태스크 그룹간의 스케일된 부하의 차이는 $2mw_{max}$보다 작거나 같다.

PROOF. ADJUST는 $\hat{L}_1 \le \hat{L}_2 \le \cdots \le \hat{L}_m$ 관계를 만족 시킨다. 그리고 같은 클러스터 내부에서 태스크 그룹간 가장 큰 스케일된 부하 차이는 항상 G_m과 G_1 사이에서 발생한다. 따라서 $\hat{L}_m - \hat{L}_1$은 다음의 식으로 나타낼 수 있다.

$$\hat{L}_m - \hat{L}_1 = (\hat{L}_m - \hat{L}_{m-1}) + (\hat{L}_{m-1} - \hat{L}_{m-2}) + \cdots + (\hat{L}_2 - \hat{L}_1)$$

$$\le 2w_{max} + 2w_{max} + \cdots + 2w_{max}$$

$$\le 2mw_{max}.$$

결론적으로, 이 lemma는 참이다. □

이어서, SVR−BASED LOAD BALANCING 알고리즘은 다음 lemma를 만족시킴을 보인다.

LEMMA 4.3. 동일 클러스터 내의 임의의 태스크 τ_i와 τ_j에 대하여, λ의 주기로 부하분산 될 때, 각 태스크의 SVR 값이 증가된 양에 대한 차이는 상수 값 이내로 제한되고, 이를 표현하면 다음과 같다.

$$-\lambda \cdot \left(\frac{1}{w_{min}} - \frac{1}{w_{min} + 2mw_{max}}\right) \leq \Delta\hat{v}_{i,j}(r) \leq \lambda \cdot \left(\frac{1}{w_{min}} - \frac{1}{w_{min} + 2mw_{max}}\right)$$

PROOF. 이를 증명하기 위하여, 먼저 $\Delta\hat{v}_i(r)$을 τ_i의 r번째 부하분산 주기인 $[(r-1)\lambda,\ r\lambda]$ 동안 발생한 SVR 증가 량으로 정의한다. SVR을 나타낸 정의에 의하여 이는 다음과 같이 표현될 수 있다.

$$\Delta\hat{v}_i(r) = \frac{1}{w(\tau_i)} \int_{(r-1)\lambda}^{r\lambda} R_i(t)\, dt \tag{19}$$

위 식의 적분으로 표현된 부분은 $R_i^{avg}(r)$ 값과 시간 $[(r-1)\lambda,\ r\lambda]$ 동안 수행된 τ_i 의 CPU 시간의 곱으로 나타낼 수 있다. 이때, 시간 $[(r-1)\lambda,\ r\lambda]$ 동안 수행된 τ_i의 CPU 시간은 스케일된 가중치에 비례하고, τ_i가 속한 실행 큐의 스케일된 부하에 반비례하므로, 식 (19)은 다음과 같이 나타낼 수 있다.

$$\Delta\hat{v}_i(r) = \frac{1}{w(\tau_i)} \cdot R_i^{avg}(r) \cdot \left(\lambda \cdot \frac{w(\tau_i)/R_i^{avg}(r)}{\hat{L}_k}\right) = \frac{\lambda}{\hat{L}_k}$$

위 식에서는 τ_i가 태스크 그룹 G_k에 속한 경우를 표현한다.

이를 이용하면, $\Delta\hat{v}_{i,j}(r)$ 다음과 같이 나타낼 수 있다.

$$\Delta\hat{v}_{i,j}(r) = \Delta\hat{v}_i(r) - \Delta\hat{v}_j(r) = \lambda \cdot \left(\frac{1}{\hat{L}_k} - \frac{1}{\hat{L}_l}\right)$$

위 식은 τ_i가 태스크 그룹 G_k에, 그리고 τ_j는 태스크 그룹 G_l에 속한 경우를 표현한 식이다.

이어서 다음 두 가지 경우를 각각 분리하여 생각한다: (a) $\Delta\hat{v}_{i,j}(r) \geq 0$ and (b) $\Delta\hat{v}_{i,j}(r) < 0$.

Case (a): $\Delta\hat{v}_{i,j}(r) \geq 0$ 인 경우.

LEMMA 4.2에 의하여 다음 식을 유도할 수 있다.

$$\Delta\hat{v}_{i,j}(r) \leq \lambda \cdot \left(\frac{1}{\hat{L}_k} - \frac{1}{\hat{L}_k + 2mw_{max}}\right) \leq \lambda \cdot \left(\frac{1}{\hat{L}_k} - \frac{1}{\hat{L}_k + 2mw_{max}}\right) \qquad (20)$$

식(20)의 최 우측 항은 $\hat{L}_k$ 값이 커짐에 따라 점차 감소한다. 따라서 $\hat{L}_k$이 최소 값을 가질 때, $\Delta\hat{v}_{i,j}(r)$ 는 최대 값을 갖게 된다. 이러한 성질을 이용하여 식 (20)을 다음과 같이 나타낼 수 있다.

$$0 \leq \Delta\hat{v}_{i,j}(r) \leq \lambda \cdot \left(\frac{1}{w_{min}} - \frac{1}{w_{min} + 2mw_{max}}\right)$$

Case (b): $\Delta\hat{v}_{i,j}(r) < 0$ 인 경우

Case (b)의 증명은 위 Case (a)와 비슷하게 이루어지며, 결론적으로 다음을 유도할 수 있다.

$$-\lambda \cdot \left(\frac{1}{w_{min}} - \frac{1}{w_{min} + 2mw_{max}}\right) \leq \Delta\hat{v}_{i,j}(r) < 0$$

Case (a)와 Case (b)를 조합하면 이 lemma가 참임을 알 수 있다. □

LEMMA 4.3을 사용하여 마지막으로, 본 연구에서 제안하는 스케줄링 기법이 동일 클러스터 내부의 임의의 두 태스크에 대하여, SVR 차이가 상수 값 이내로 제한됨을 증명한다.

THEOREM 1. 동일 클러스터 내의 임의의 두 태스크 τ_i와 τ_j에 대하여, $|\hat{v}_{i,j}(t)|$ 값은 상수 값 이내로 제한되며, 이를 식으로 나타내면 다음과 같다.

$$|\hat{v}_{i,j}(t)| \leq C\lambda \quad \text{where } C = \left(\frac{1}{w_{min}} - \frac{1}{w_{min} + 2mw_{max}}\right)$$

PROOF. $|\hat{v}_{i,j}(t)|$ 값이 가장 클 때는 매 부하분산 주기마다 발생한다. 따라서 본 증명 과정에서는 시간 $t = r\lambda$의 경우만 고려한다. 또한, 이를 증명하기 위하여 본 학위논문에서는 수학적 귀납법을 사용한다.

먼저 $t = 0$인 경우를 생각한다. 모든 태스크의 SVR 값은 t =0일 때 0이다. 따라서 이는 theorem을 만족한다.

다음으로 $t = r\lambda$일 때 theorem을 만족하면, $t = (r+1)\lambda$일 때도 만족함을 보인다. $t = (r+1)\lambda$일 때, 두 태스크의 SVR 값 차이는 다음과 같다.

$$\hat{v}_{i,j}(r+1)\lambda = \hat{v}_{i,j}(r\lambda) + \Delta\hat{v}_{i,j}(r)$$

먼저 $\hat{v}_{i,j}(r\lambda) < 0$인 경우를 생각한다. 이 경우, $\Delta\hat{v}_{i,j}(r)$ 값이 양의 값이고, $C\lambda$보다 작음을 뜻한다. 이는 LEMMA 4.3의 Case (a)를 통해서 쉽게 알 수 있다. 귀납법의 유도 가정(induction hypothesis)에 의하여 $\hat{v}_{i,j}(r\lambda) \geq -C\lambda$ 는 참이다. 그리고 $\hat{v}_{i,j}(r+1)\lambda$ 는 $\hat{v}_{i,j}(r\lambda)$ 보다 크다. 따라서 $-C\lambda \leq \hat{v}_{i,j}(r+1)\lambda \leq C\lambda$을 만족한다.

반대로, $\hat{v}_{i,j}(r\lambda) \geq 0$인 경우를 생각한다. 이는 $\Delta\hat{v}_{i,j}(r)$값이 음의 값이고, $-C\lambda$보다 큰 값임을 뜻한다. 이는 LEMMA 4.3의 Case (b)를 통해서 알 수 있다. $t = r\lambda$에서 이 theorem이 참임을 가정하였기 때문에, $\hat{v}_{i,j}(r\lambda) \leq C\lambda$ 는 참이다. $\hat{v}_{i,j}(r+1)\lambda$ 값은 $\hat{v}_{i,j}(r\lambda)$ 보다 작다. 따라서 $-C\lambda \leq \hat{v}_{i,j}(r+1)\lambda \leq C\lambda$을 만족한다.

결론적으로, $\hat{v}_{i,j}(r\lambda) < 0$ 인 경우와 $\hat{v}_{i,j}(r\lambda) > 0$ 인 경우 모두, $-C\lambda \leq \hat{v}_{i,j}(r+1)\lambda \leq C\lambda$을 만족함을 알 수 있다. 따라서 이 theorem 은 참이다. □

다음으로, 본 연구에서 제안하는 공정할당 스케줄링 기법의 시간 복잡도를 분석한다. 본 연구에서 대상으로 하는 클러스터는 앞서 설명한대로, n개의 태스크와 m개의 코어로 이루어진 클러스터이다. ALGORITHM 2에 기술된 SVR-BASED LOAD BALANCING 은 세 개의 서브루틴을 가지고 있다: SORT, SPLIT, ADJUST.

SORT 에서는 merge-sort를 수행하므로, 이의 시간 복잡도는 직관적으로 $O(n \log n)$임을 알 수 있다. SPLIT 이 수행되는 동안 각각의 태스크 그룹은 서로 다른 태스크들을 가지고 있다. 따라서 SPLIT 의 시간 복잡도는 $O(n)$ 이다.

다음으로 ADJUST 를 살펴본다. 독립적으로 CHECKANDMOVETASKS 함수가 호출될 때, 이는 $O(1)$의 시간 복잡도를 갖는다. 이 함수가 호출될

때, 최대 태스크 이주 발생 수는 $\frac{w_{max}}{w_{min}}$보다 작다. 그리고 전체적으로 이 함수는 $O(n)$번 불릴 수 있다. 따라서 ADJUST 는 $O(mn)$의 시간 복잡도를 갖는다. 전체적으로 보면, 본 연구에서 제안된 스케줄링 기법의 시간 복잡도는 $O(n\log n + n + mn) \approx O(nlog\ n)$이다.

제 5 장 실험 및 검증

제안된 각각의 스케줄링 기법의 실효성을 평가하기 위해서, 본 장은 수행한 실험에 대하여 설명한다. 실험 대상으로는 비대칭 멀티코어 아키텍처를 사용하는 상용 제품을 사용하였다. 본 장은 다음의 세 개의 절로 구성되어 있다. 1절은 비대칭 멀티코어 아키텍처의 코어 타입 선택 방식과 전체 코어 사용 방식을 나타내는 하드웨어적 환경을 기술한다. 또한, 이에 사용된 소프트웨어적 명세에 대하여 설명한다. 2절은 정량적으로, 제안된 기법의 실효성을 평가하기 위한 성능 지표들을 기술한다. 마지막으로, 3절에서는 구현된 기법들을 적용한 실험적 결과들을 검증한다.

제 1 절 실험 환경

본 연구의 목표는 비대칭 멀티코어 아키텍처가 나타내는 두 가지 모드에 최적화된 스케줄링 기법을 제안하는 것이다. 이를 위해서, 본 연구에서 제안된 스케줄링 기법들은 각 모드를 하드웨어적으로 지원하고 있는 상용 제품에 구현되었다. 구체적으로, 빅리틀 아키텍처의 CPU간 이주 모드에 적합한 스케줄링 기법은 Android 스마트폰 Galaxy S4위에 구현되었다. 또한, GTS 모드에 적합한 스케줄링 기법은 ARM사의 Versatile Express TC2 board에 구현되었다 [5][9]. 이들 대상 시스템에 대한 소프트웨어적 및 하드웨어적 자세한 구성 요소들은 표 4에 명시하였다.

표 4. 대상 시스템의 하드웨어 및 소프트웨어적 명세

구분			설명
비대칭 멀티코어 아키텍처용 저전력 스케줄링	HW	보드 명	Galaxy S4
		빅코어	Cortex-A15X4
		리틀코어	Cortex-A7X4
		메모리	2GB SDRAM
	SW	Android	Android version 4.1.1
		Linux	kernel version 3.4.0
비대칭 멀티코어 아키텍처용 공정할당 스케줄링	HW	보드 명	Versatile Express TC2
		빅코어	Cortex-A15X2
		리틀코어	Cortex-A7X3
		메모리	2GB DDR2
	SW	Android	Android version 4.4.2
		Linux	kernel version 3.10.54

제 2 절 실험 시나리오 및 성능 지표

본 절에서는 본 연구에서 수행했던 실험들이 어떤 테스트 시나리오와 workload를 사용했는지에 대하여 기술한다. 또한, 본 학위논문에서 제안된 스케줄링 기법들을 표 4에 나타낸 대상 시스템들에 구현했을 때, 최적화 달성 여부를 평가할 수 있는 정량화된 성능 지표를 정의한다. 본 연

구에서, 최적화 이슈는 (1) 빅리틀 아키텍처의 CPU간 이주 모드에서의 저전력 특성에 대한 최적화 달성 여부, 그리고 (2) GTS 모드에서의 공정할당성 최적화 달성 여부이다.

2.1 저전력 스케줄링 기법 최적화

본 연구에서 제안된 빅리틀 아키텍처의 CPU간 이주 모드용 스케줄링 기법이 최적화를 달성했는지 확인하기 위하여, 두 가지 종류의 실험을 수행하였다. 첫째, 벤치마크를 통한 저전력 특성을 체크하였다. 사용된 벤치마크들은 생성과 소멸을 반복한다. 따라서, 벤치마크들의 생성/소멸 비율을 조절하면서 여러 가지 서로 다른 실험 상황을 만들어 낸다. 둘째, 소프트웨어로 디코딩되는 비디오 플레이백을 사용하였다. 이 실험을 통하여, 실제 생활에서 사용되는 Android 응용프로그램들이 QoS를 해치지 않으면서 에너지 효율적으로 운용되는지 확인하였다.

본 연구에서 사용한 벤치마크는 SPEC CPU2006을 사용하였다. 이는 여러 종류의 CPU-intensive 벤치마크들로 이루어져 있으며, 각각의 벤치마크들은 모두 한 개의 태스크로 이루어져 있다. 이 중, 본 학위논문에서는 h264ref 와 gcc를 사용하였고, 이들에 대한 설명은 표 5에 나타내었다.[45][46]

이 벤치마크들의 특성은 실제 Android 응용프로그램과는 다르다. 이들은 일단 시작되면 실행을 마칠 때까지 Sleep 상태로 가거나, IO 동작에 의하여 블로킹되지 않는다. 따라서, 이들 벤치마크들은 시스템의 준비 큐(wait queue)로 진입하지 않고, 수행되는 동아 단지 부하분산 정책에 의하여 실행 큐(run-queue) 사이만 옮겨 다니게 된다.

표 5. 저전력 스케줄링 기법에 사용된 SPEC CPU2006 벤치마크

Test suites	Description
h.264ref	This test suite is a reference implementation of H.264/AVC video compression. It encodes input video data.
gcc	This test suite is based on gcc version 3.2. It generates code for AMD Opteron processor. It runs as a compiler with many of its optimization flgs enabled

하지만, 실제 Android 단말에서는, 어떤 한 응용프로그램이나 태스크가 특정 코어를 독점하는 현상을 막기 위하여, 준비 큐로 자주 태스크들이 옮겨가고, 다시 실행 큐로 할당이 된다. 본 연구에서 제안하는 스케줄링 기법은, 이러한 실제 Android 단말에서 사용되는 응용프로그램들을 운용함에 있어서 저전력효과를 나타내는데 목적이 있다. 따라서, 단순히 CPU-intensive 벤치마크들을 실험에 사용하면, 실제 Android 응용프로그램들의 동작 시나리오를 적절히 반영하지 못한다.

따라서, 본 연구에서 제안된 스케줄링 기법을 테스트하기 위하여, 그림 18에 보이는 것처럼 실험 시나리오를 구성하였다. 실험 시나리오에서는 SPEC CPU 2006에서 제공하는 h264ref와 gcc 두 개의 벤치마크를 사용하였다.

h264ref의 encoding 프레임 수는 2로 설정하였고, gcc가 컴파일하는

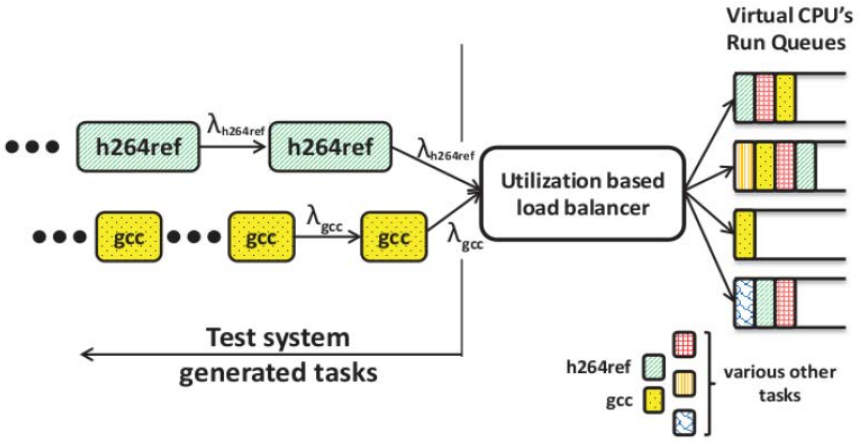


그림 18. h264ref 와 gcc 벤치마크를 사용한 태스크의 생성/소멸 반복

데 사용하는 파일의 크기는 3.4KB로 하였다. h264ref 태스크들은 그림의 사용률 기반 부하분산기(utilization based load balancer) 앞에 고정된 도착률(1/3 arrivals/second)로 도착한다.

일단 h264ref 태스크가 부하분산기에 의하여 가상 CPU에 할당이 되면, 계속 수행되다가 시스템에서 빠져나간다. 하지만 동일한 태스크는 다시 생성되어 시스템에 진입하게 된다. 따라서, 이러한 실험 환경은 태스크가 가상 CPU에서 실행을 멈추고 준비 큐로 진입 후, 다시 실행 큐 할당을 기다리는 상황과 동일한 환경을 제공한다.

본 실험에서는 $\lambda_{h264ref}$ 를 h264ref의 도착률, gcc의 도착률은 λ_{gcc} 로 표현하였다. 여기에서 도착률이란 새로운 h264ref와 gcc 태스크가 생성되어 부하분산기 앞에 도착하는 비율을 나타낸다. 즉, 다시 말하면 1초동안 몇 개의 태스크가 생성되는지를 나타낸다. 이 두 도착률은 실험을 수행하는 동안 변하지 않고 고정된다. 본 연구에서는 $\lambda_{h264ref}$ 와 달리 λ_{gcc} 는 여러 가지 값을 취하여 각각 실험하였다. 즉, [20 arrivals/second, 80

arrivals/second] 사이의 숫자를 취하게 하였으며, 그 간격은 10 arrivals/second으로 하였다.

실험에서 측정 시, 각 가상 CPU당 평균 gcc의 서비스률 μ_{gcc}는 14.3 tasks/second 혹은, 약 70ms/task 였다. 따라서, 이는 시스템이 57.2 gcc tasks/seconds를 운용할 수 있다는 뜻이다. 도착률이 50 arrivals/second 혹은, 이보다 작을 때, 태스크들은 wait → run → wait loop 루틴을 반복하고, 이때 어떤 고정된 값의 wait 시간을 갖는다. 도착률이 60 arrivals/second 혹은 이보다 큰 값을 가질 때는, 앞서 생성된 태스크가 그 실행을 마치기 전에 새로운 태스크가 생성되어 시스템으로 유입됨을 알 수 있었다.

이렇게 태스크의 도착률을 변화시킴으로써, 태스크가 서로 다른 wait 시간을 가지면서, 마치 시스템의 준비 큐에 머물게 하는 효과를 만들었다. 또한, 이는 실제 Android의 응용프로그램들이 준비 큐에 머물다가 실행 큐를 선택해서 진입하는 시나리오를 재연하였다. 이 실험 방법은 벤치마크에 인위적인 wait 상태를 부과하되, SPEC CPU2006 벤치마크를 수정하지 않고 사용할 수 있게 하였다.

SPEC CPU2006 벤치마크를 수행한 후 저전력 성능 지표로서, 소비된 에너지(mAh)를 측정하였다. 이는 gcc 태스크간 도착률 λ_{gcc} 을 [20 arrivals/second, 80 arrivals/second] 사이의 값을 갖게 하면서 측정하였다. 이와 동시에, 각 λgcc 마다 gcc가 수행을 완료한 시간을 측정하였다. 이는 에너지소비와 성능의 관계에 있어서, 제안된 스케줄링 기법이 얼마 만큼의 성능을 감소시키면서 에너지소비를 줄이는지 확인하기 위해서이다.

두 번째 실험으로, 하드웨어 디코더를 쓰지 않고, 100% 소프트웨어로 디코딩되는 Android MX player를 실험 대상으로 사용하였다. MX player

는 21개의 태스크들로 이루어진 Android 응용프로그램이다. 이들 21개의 태스크들은 가상 CPU에서 수행된 후, Sleep과 같은 블로킹 상태를 만나서 시스템의 준비 큐에 옮겨진다. 또한 이들은 부하분산 정책에 의하여 실행 큐들 사이를 옮겨 다닌다.

이와 같은 전형적인 Android 응용프로그램의 특성을 가진 MX player를 본 연구에서 활용함으로써, 제안된 기법의 저전력 최적화 달성 여부를 판단할 수 있다. 실험 대상으로 사용한 디코딩용 파일은 h.264 codec으로 압축된 1920X1080 full HD 비디오 clip이다. 본 실험에서 사용된 이 비디오 clip은 31초 동안 디코딩하여 재생시키는 콘텐츠이다.

MX player를 수행한 후 저전력 성능 지표로서, 소비된 에너지(mAh)를 측정하였다. 이는 31초동안 비디오 clip을 재상 하면서 소비된 에너지이다. 이와 동시에, FPS(Frame per Second)를 1초마다 측정하고, 평균값을 측정하였다. 이는 에너지소비와 QoS의 관계에 있어서, 제안된 스케줄링 기법이 얼마 만큼 Android 응용프로그램의 QoS 특성을 해치면서 에너지소비를 줄이는지 확인하기 위해서이다.

2.2 공정할당 스케줄링 기법 최적화

본 학위논문에서 제안하는 공정할당 스케줄링 기법의 최적화 달성 여부를 확인하기 위해서, 다음의 세 가지 실험을 수행하였다: (1) 태스크 간 $\hat{v}_{max}(t)$를 측정, (2) 동일한 태스크를 여러 개 수행시킨 후 각 태스크들의 완료시간 편차를 측정, (3) 제안된 기법으로 인하여 발생하는 런타임 오버헤드 측정. 열거한 세 가지 실험에서 부하분산 주기 λ는 기존의 CFS와 동일하게 1초로 하였다.

첫째, 본 연구에서 제안한 공정할당 스케줄링 기법을 적용하였을 때와 하지 않았을 때의 $\hat{v}_{max}(t)$를 측정하였다. 제안된 기법의 실효성을 검증하기 위해서, 시스템 운용시간 증가에 따른 $\hat{v}_{max}(t)$ 변화 값을 비교 하였다. 4.1절에서 먼저 기술하였듯이, 이상적인 공정할당 스케줄러는 $\hat{v}_{max}(t)$를 0으로 유지시킨다. 따라서, 본 실험에서는 시간이 증가하여도 이 값이 발산하지 않고 상수 값 이내로 제한됨을 검증하고자 한다.

시간에 따른 $\hat{v}_{max}(t)$ 값 변화를 측정하기 위해서, 한 개의 태스크로 구성된 응용프로그램과 여러 개의 태스크로 구성된 응용프로그램을 사용하였다. 이를 위해서 SPEC CPU2006 과 PARSEC 벤치마크를 각각을 나타내는 부하로 사용하였다 [25][26][45][46]. SPEC CPU2006 벤치마크에서는 bzip2, bwaves, mcf 등을 사용하고, PARSEC 벤치마크 중 swaptions, blacksholes, ferret 등을 실험에 사용하였다. 그리고 이들에 대한 설명을 표 6, 7에 나타내었다. SPEC CPU2006 벤치마크를 사용시

표 6. 공정할당 스케줄링 기법에 사용된 SPEC CPU2006 벤치마크

Test suites	Description
bzip2	This test suite is based on an open-source file compression program named bzip2. Each input is compressed and decompressed.
bwaves	This test suite is a floating point benchmark. It numerically simulates blast waves in three dimensional transonic transient laminar viscous flow.
mcf	This test suite is a program used for single-depot vehicle scheduling in public mass transportation.

에는 동일한 16개의 벤치마크를 시스템에 부가하였고, PARSEC 벤치마크에서는 여러 개의 태스크로 이루어진 한 개의 벤치마크를 사용하였다. 본 실험에서는 $\hat{v}_{max}(t)$ 값을 초단위로 측정하여 성능 지표로 사용하였다.

둘째, $\hat{v}_{max}(t)$이 발산하지 않고 상수 값 이내로 제한되는 것이 공정할당에 어떤 영향을 주는지, 보다 직관적으로 확인하였다. 이를 위하여, 본 실험에서는 동일한 workload를 여러 개 복사해서 시스템에 부가하였다. 해당 workload는 한 개의 태스크로 구성된 응용프로그램이며, 이는 CPU-intensive한 bubble-sort를 수행하고 정해진 시간 동안 sleep하는 간단한 동작을 반복적으로 수행한다.

표 7. 공정할당 스케줄링 기법에 사용된 PARSEC 벤치마크

Test suites	Description
blackscholes	This test suite runs a computational finance application which calculates the prices for a portfolio via the BlackScholes partial differential equation (PDE). It is the simplest of all PARSEC workload.
swaptions	This test suite runs a computational finance application which employs a Monte Carlos simulation to analyze non Markovian models.
ferret	This test suite is a This application is based on the Ferret toolkit which is used for content-based similarity search of feature-rich data such as audio, images, video, 3D shapesprogram used for single-depot vehicle scheduling in public mass transportation.

본 실험에서 CPU-intensive하고 한 개의 태스크로 구성된 SPEC CPU2006을 사용하지 않고, 앞서 설명한 인위적인 응용프로그램을 사용한 이유는 다음과 같다. 저전력 스케줄링 기법 최적화 달성 여부를 확인하는 실험에서 설명되었듯이, SPEC CPU2006 벤치마크의 태스크들은 일반적인 응용프로그램과 달리 시스템의 준비 큐로 이동하지 않고, 코어와 실행 큐에만 존재한다. 따라서 본 실험에서도 실제상황에 가까운 workload를 나타내기 위해서 실행 큐와 준비 큐를 옮겨 다니는 부하를 사용하였다.

성능 지표로서는 각 응용프로그램들의 완료시간을 사용하였다. 즉, 이상적인 공정할당 스케줄링 기법을 사용할 경우, 각각의 동일 태스크들은 같은 시점에 완료되어야 한다. 따라서, 본 실험에서는, 각 태스크들이 나타내는 완료시간의 표준편차, 완료시간의 최대/최소 값의 차이 등을 측정하였다.

셋째, 제안된 공정할당 스케줄링 기법이 런타임에 나타내는 오버헤드를 측정하였다. 이를 위해서, 앞서 사용했던 SPEC CPU2006 과 PARSEC 벤치마크를 사용하였다. 런타임 오버헤드를 파악하기 위하여, 제안된 기법이 적용되었을 경우, 벤치마크 완료시간이 적용되기 전보다 얼마나 늘어났는지를 측정하였다.

제 3 절 실험적 검증 결과

본 절에서는 빅리틀 아키텍처의 (1) CPU간 이주 모드용 저전력 스케줄링 기법과 (2) GTS 모드용 공정할당 스케줄링 기법에 대한 최적화 달성 여부를 실험적 검증 내용을 통하여 기술한다. 3.1절에서는 저전력 최적화 특성을 벤치마크 수행 시 감소된 에너지 사용으로써 확인한다. 또한,

Android 응용프로그램을 수행 시 에너지 소비가 감소하는 지를 살펴본다. 마지막으로, 제안된 기법으로 인한 런타임 오버헤드를 확인하기 위하여, 벤치마크 수행시간을 비교하고, Android 응용프로그램의 QoS 특성 저하 여부를 확인한다.

3.2 절에서는 공정할당 스케줄링 기법의 실험적 검증 검증 결과를 태스크 간 최대 SVR 차이가 상수 값 이내로 제한됨을 통하여 확인한다. 또한, 이러한 결과가 여러 개의 동일한 태스크들 수행 시, 그들의 완료 시간 편차가 감소하는 결과로 나타남을 검증한다. 마지막으로, 제안한 기법들로 인하여 발생하는 런타임 오버헤드에 대하여 설명한다.

3.1 저전력 스케줄링 기법의 실험 결과

■ **벤치마크 테스트**

그림 19는 gcc 벤치마크의 도착률 λ_{gcc}를 [20 arrivals/second, 80 arrivals/second] 사이의 값을 취하면서 측정한 에너지 소비(mWh) 결과를 나타낸다. 그림에서, Legacy의 의미는 본래의 CFS에서의 결과를 나타내고, $E_A(U)$와 $E_B(U)$는 각각 3장에서 설명한 사용률 기반 추정기와 수행이력을 반영한 사용률 추정기를 사용했을 때의 결과를 나타낸다. 그림에 나타낸 세 개의 직선은, λ_{gcc}를 변화시키며 기록한 실험 결과를 Linear regression을 사용하여 나타낸 결과이다.

그림에서 알 수 있듯이, $E_A(U)$ 혹은 $E_B(U)$ 추정기를 사용하는, 본 연구에서 제안한 스케줄링 기법이 에너지 소비를 감소시켰다. Linear regression이 나타내는 라인을 보았을 때, λ_{gcc} 값이 커질 때 즉, 시스템에 부하가 더 많아질 때 소비된 에너지 차이는 더 커짐을 알 수 있다.

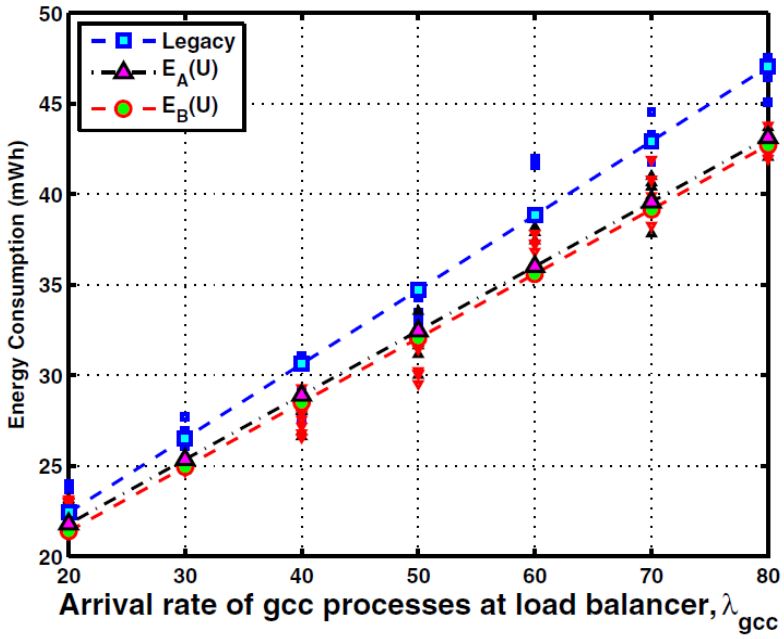


그림 19. gcc 도착률에 따른 에너지 소비 비교

표 8은 그림 19에 나타낸 λ_{gcc} 값들 중, 20, 30, 60 arrivals/second의 도착률일 때 소비된 에너지를 구체적인 값으로 나타내고 있다. 또한, 표에서는 제안된 스케줄링 기법에 의하여 발생한 런타임 오버헤드를 나타내고 있다. 표에서 나타낸 Legacy, $E_A(U)$, $E_B(U)$의 의미는 앞서 그림 19에서 설명한 바와 동일하다. $\Delta_A\%$ 와 $\Delta_B\%$는 각각 본래의 CFS와 $E_A(U)$ 및 $E_B(U)$가 나타내는 값을 비교했을 때, 측정된 값의 변화율을 나타낸다.

표에서 알 수 있듯이, 소비 전류, 전력, 에너지 모든 측면에서 제안된 스케줄링 기법이 더 작은 값을 나타내었다. 또한, 사용률 기반 추정기를 사용했을 때보다 수행이력을 반영한 사용률 추정기를 사용했을 때, 저전

력 효과가 더 큼을 알 수 있었다. 본 학위 논문에서 제안한 두 추정기의 차이점은 다음과 같다.

수행이력을 반영한 사용률 추정기는, 어떤 태스크가 준비 큐에서 빠져

표 8. gcc 도착률에 따른 에너지 소비 및 런타임 오버헤드

λ_{gcc}	구분	Legacy	$E_A(U)$	Δ_A%	$E_B(U)$	Δ_B%
20	Time (s)	36.40	36.44	0.12%	36.67	0.75%
	Current (mA)	591.87	577.74		570.06	
	Power (mW)	2359.81	2303.63		2272.79	
	Energy (mWh)	23.86	23.31	−2.29%	23.15	−2.96%
30	Time (s)	38.65	39.66	2.62%	39.84	3.09%
	Current (mA)	625.26	573.90		563.04	
	Power (mW)	2493.04	2288.24		2244.89	
	Energy (mWh)	26.77	25.21	−5.83%	24.84	−7.18%
60	Time (s)	50.88	52.84	3.84%	52.54	3.25%
	Current (mA)	743.24	636.81		638.29	
	Power (mW)	2963.27	2539.00		2544.80	
	Energy (mWh)	41.88	37.26	−11.04%	37.13	−11.35%

나와 실행 큐로 진입할 때, 그 태스크가 가상 CPU 사용률을 변화 시키는 정도를 예측한 후 실행 큐를 결정한다. 하지만 사용률 기반 추정기는, 태스크가 어떠한 실행 큐로 진입하더라도, 모든 가상 CPU들의 사용률은 진입 전과 동일하다는 가정하에 실행 큐를 결정한다. 즉, 현재 시점의 가상 CPU 사용률만 보고 판단한다. 따라서, 제안한 스케줄링 기법의 사용률 예측 방법이 실효성이 있다는 것을 검증하고 있다.

표에 나타낸 Time (s)은 gcc 벤치마크의 완료시간 초단위로 나타낸 결과이다. Legacy와 두 가지 제안된 추정기를 사용했을 때를 비교하면, 0.12% ~ 3.84% 수행시간 증가를 나타내었다. 즉 이는 런타임 오버헤드를 나타내며, 시스템의 부하가 커지더라도, 아주 작은 폭으로 증가함을 알 수 있었다.

■ Android 응용프로그램 테스트

벤치마크를 활용한 검증에 부가하여, 실제 Android 응용프로그램에서도 제안된 저전력 스케줄링 기법이 실효성이 있는지 확인하였다. 이를 위하여 소프트웨어로 디코딩되는 Android MX player를 사용하였다.

표 9는 이의 결과를 나타내고 있다. 이 표에 나타낸 Legacy, $E_A(U)$, $E_B(U)$의 의미는 벤치마크 테스트 결과를 설명할 때 기술한 바와 같다. 실험에서, 완전한 frame rate를 나타내기 까지, Legacy, $E_A(U)$, $E_B(U)$ 모두 2초간의 준비 시간이 필요했다. 또한, MX player가 실행을 마치는 시점의 마지막 2초 역시, 세가지 실험 환경 모두 완전한 frame rate를 나타내지 못했다. 따라서 본 실험에서, 시작 시점과 마지막 시점의 총 4초는 QoS 지표인 평균 FPS 계산시 제외시켰다. 하지만 다른 측정 결과에는 해당 4초동안의 데이터를 반영하였다.

표 9. 소프트웨어로 디코딩된 MX player 실험 결과

	Legacy-CPU	$E_A(U)$	$E_B(U)$
Time (s)	31.53	31.39	31.43
Current (mA)	690.78	650.46	644.28
Power (mW)	2754.34	2593.50	2568.94
Energy (mWh)	24.12	22.61	22.43
Energy Δ%	-	-6.68%	-7.53 %
Mean FPS	30.0	29.964	30
σ_{FPS}	0	0.429	0.385

표 9에서 알 수 있듯이, 소비 전류, 전력, 에너지 모든 측면에서 제안된 스케줄링 기법이 더 작은 값을 나타내었다. 특히, 수행이력을 반영한 사용률 추정기($E_B(U)$로 표시)를 사용했을 때, 앞서 기술한 벤치마크 테스트 결과와 마찬가지로, 더 큰 에너지 감소를 나타내었다. 이러한 결과로 인하여, 실제 Android 응용프로그램에서도 제안된 스케줄링 기법의 사용률 예측 방법이 실효성이 있다는 것을 검증하고 있다

제안된 스케줄링 기법이 침해할 수 있는 QoS 특성을 파악하기 위하여, 평균 FPS를 측정하였다. 표에서 알 수 있듯이, $E_B(U)$로 표시된, 수행이력을 반영한 사용률 추정기를 사용했을 때는 QoS 특성 침해가 없었다. 또한, $E_A(U)$로 표시한 사용률 기반 추정기 사용시에도, 0.1%의 사용자가

체감할 수 없는 정도의 작은 성능 손실이 관찰되었다.

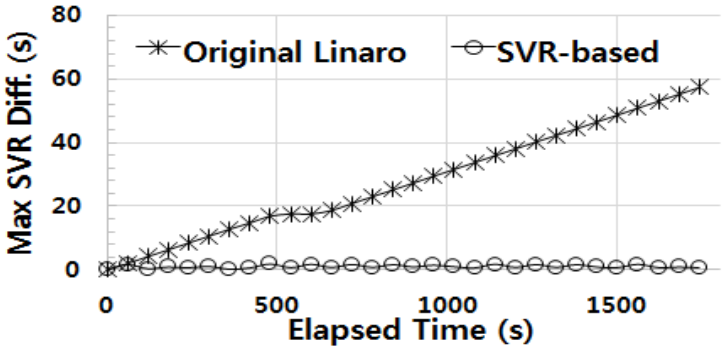
3.2 공정할당 스케줄링 기법의 실험 결과

■ **태스크간 SVR 차이 측정**

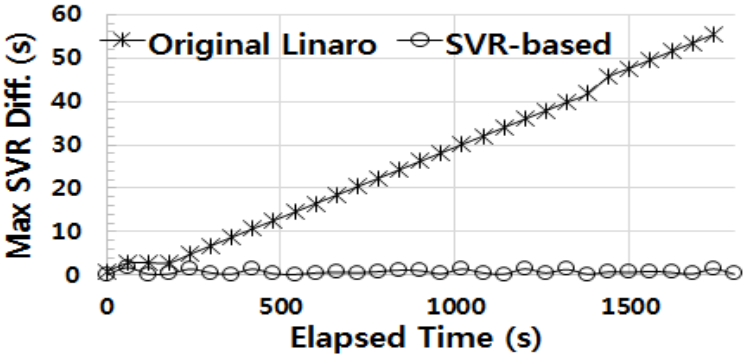
제안된 공정할당 스케줄링 기법이 빅리틀 아키텍처의 GTS 모드에서 최적화를 달성하는지 확인하기 위해서, 먼저 태스크 간의 $\hat{v}_{max}(t)$ 값을 본 학위논문에서 제안한 기법을 적용했을 때와 적용하지 않았을 때를 비교하였다.

그림 20는 한 개의 태스크로 구성된 bzip2를 16개 복사해서 시스템에 부가한 결과이고, 그림 21은 swaptions 한 개를 16개의 태스크로 분산시켜 수행했을 때 결과를 나타낸다. 본 학위논문에서 제안하는 스케줄링 기법의 핵심은 두 개의 클러스터 내부에서 각각 $\hat{v}_{max}(t)$를 상수 값 이내로 제한시키는 것이다. 따라서 본 실험에서는 각 클러스터별 $\hat{v}_{max}(t)$를 측정한다. 이어 클러스터 구분 없이 시스템 전체적으로 태스크 간 $\hat{v}_{max}(t)$를 측정하여, 클러스터 별로 $\hat{v}_{max}(t)$값을 상수 값 이내로 제한함이 시스템 전체적으로 어떤 영향을 미치는지 확인하였다.

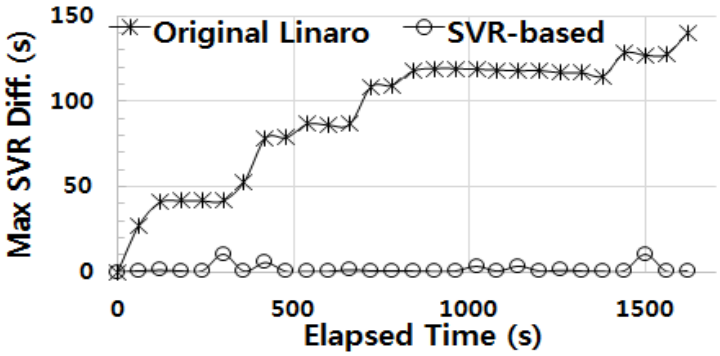
그림 20과 그림 21의 (a)는 빅클러스터에서 $\hat{v}_{max}(t)$를 측정한 결과이고, 그림 20과 그림 21의 (b)는 리틀클러스터에서 $\hat{v}_{max}(t)$를 측정한 결과이다. 그림에서 알 수 있듯이, 클러스터 타입과 벤치마크 종류에 관계없이 제안된 스케줄링 기법은 $\hat{v}_{max}(t)$ 값을 상수 값 이내로 제한시키고 있다. 하지만, 본래의 Linaro 스케줄링 프레임웍의 결과에서는 $\hat{v}_{max}(t)$ 값이 실험 시간이 증가함에 따라 발산함을 알 수 있다.



(a)

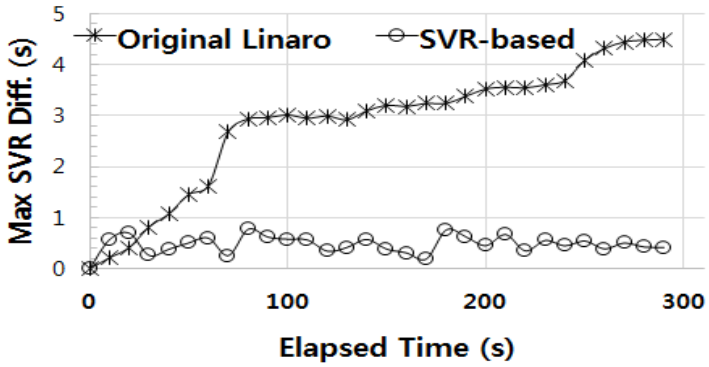


(b)

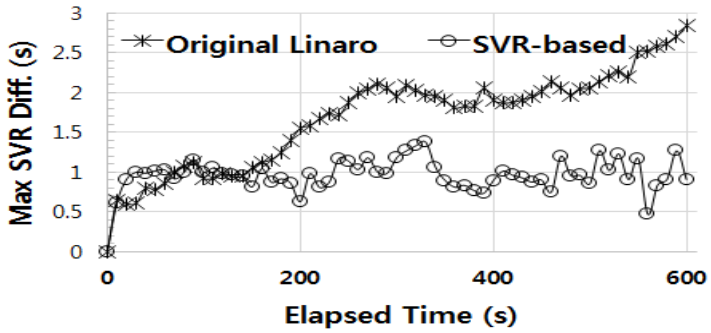


(c)

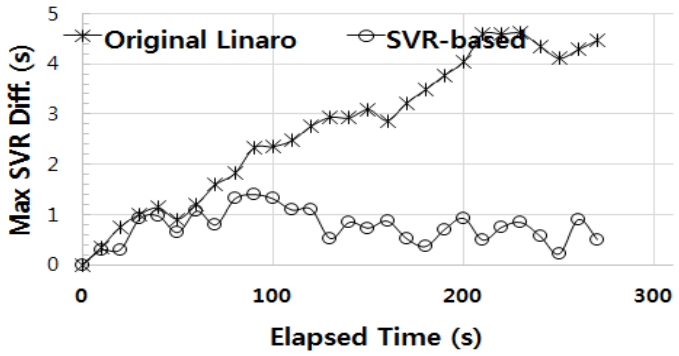
그림 20. SPEC CPU2006 (bzip2)를 사용한 최대 SVR 차이



(a)



(b)



(c)

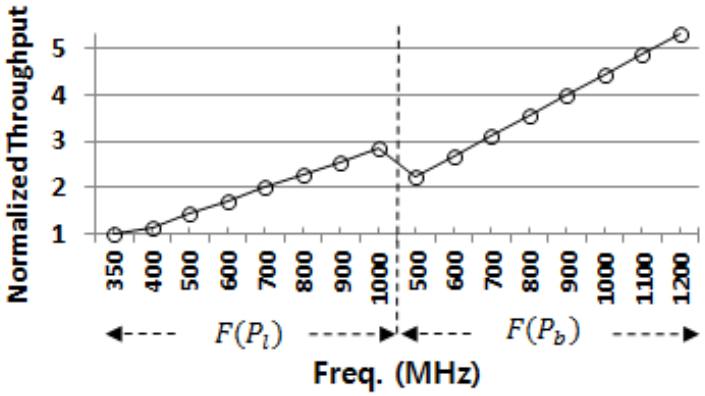
그림 21. PARSEC (swaptions)를 사용한 최대 SVR 차이

그림 20와 그림 21의 (c)는 클러스터를 구분하지 않고 시스템 전체의 $\hat{v}_{max}(t)$ 값을 측정한 결과이다. 본래의 Linaro 스케줄링 프레임웍을 적용할 경우, 앞서 설명한 클러스터별 결과와 동일하게 시간이 지남에 따라 계속 $\hat{v}_{max}(t)$ 값이 발산함을 알 수 있다. 반면에, 본 연구에서 제안한 스케줄링 기법을 적용 시, 동일한 16개의 벤치마크 테스트에서는 10.3초, 16개의 태스크로 이루어진 한 개의 벤치마크 테스트에서는 1.4초로 제한됨을 알 수 있었다.

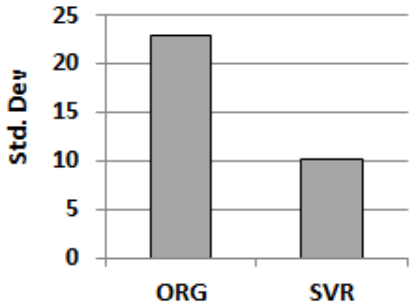
■ **동일 workload 테스트**

다음으로, 유한한 양의 CPU-intensive한 일을 수행하고, 한 개의 태스크로 이루어진 workload를 여러 개 복사하였다. 이어서, 이들을 시스템에 부가한 후, 이들의 완료 시간 편차를 측정하였다. 그림 22의 (a)는 이 workload의 각 f'에 따른 성능을 나타내고 있다. 이를 위하여 offline에서, 사용 가능한 f' 값들을 고정 시킨 후 각각의 성능을 측정하였다. 이때 $f' = f'_{min}$ 인 경우의 측정된 성능을 baseline으로 설정하고, 나머지 각각의 f'에서 측정된 성능들을 baseline에 대한 배수로 설정하였다. 그림에서 f'_{min}는 350 MHz를 나타낸다.

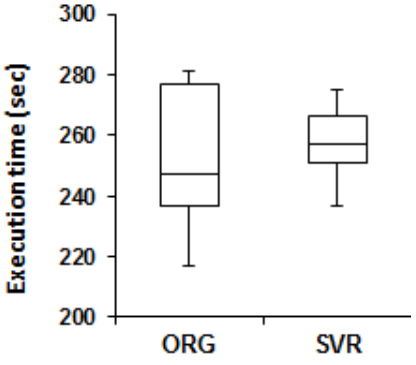
그림 22에서 ORG는 본래의 Linaro 스케줄링 프레임웍의 결과를 나타내고, SVR은 본 연구에서 제안된 스케줄링 기법의 결과를 나타낸다. 그림의 (b)는 동일한 16개 workload의 완료시간 표준편차를 나타내고 있다. 그림에서 알 수 있듯이, 제안된 스케줄링 기법이 원래의 Linaro 스케줄링 프레임웍보다 표준편차가 56% 작았다. 그림의 (c)는 동일 workload들 각각의 완료시간 분포를 나타내고 있다. 그림에서 보이듯이, 완료시간 최대편차 역시 제안된 스케줄링 기법이 49.6% 더 작은 값을 나타내었다.



(a)



(b)



(c)

그림 22. 16개 동일 workload의 완료시간 편차

벤치마크 및 동일 workload 를 사용한 실험들을 통하여, 제안된 공정할당 스케줄링 기법이 비대칭 멀티코어 아키텍처에서 향상된 공정할당성을 나타냄을 쉽게 알 수 있다.

■ **런타임 오버헤드 측정**

마지막으로 공정할당 스케줄링 프레임웍이 야기하는 오버헤드를 설명한다. 제안된 기법으로 인한 오버헤드는 기존의 Linaro 스케줄링 프레임웍에 추가로 탑재되거나 수정된 소프트웨어 모듈로 인한 지연시간으로 정의한다. 이러한 지연시간의 원인이 될 수 있는 점들을 다음과 같이 설명할 수 있다.

본 연구에서는, 기존의 virtual runtime 기반의 per-core 스케줄러를 SVR(scaled virtual runtime) 기반으로 변경하였다. 그리고 태스크 별 SVR 값을 구하기 위하여 SVR calculator를 추가하였다. 이에 매 스케줄링 tick마다 발생하는 SVR calculator의 OLS regression 동작이 지연시간을 초래할 수 있다.

태스크간 $\hat{v}_{max}(t)$를 상수 값 이내로 제한하기 위하여, 기존의 가중치 기반 부하분산 정책을 SVR 기반 부하분산으로 변경하였다. 기존의 가중치 기반 부하분산 정책은 실행 큐간 부하의 차이가 허락된 값 이내로 들어오는지 체크한 후 태스크를 옮긴다. 이에 반하여 제안된 스케줄링 기법은 SORT, SPLIT, ADJUST 등의 서브루틴을 수행하면서, merge-sort 및 태스크 그룹핑 작업을 수행한다. 이렇게 변경된 부하분산 방식이 지연요소로 나타날 수 있다.

이러한 지연시간을 발생시킬 수 있는 점들을 확인하기 위하여 전체론적인 관점에서 런타임 오버헤드를 측정하였다. 다시 말하면, 런타임 오버헤드로 인하여 태스크들이 지연되어 완료하는 점을 활용한다.

이를 위하여, 한 개의 태스크로 이루어진 bzip2, bwaves, mcf 등의 SPEC CPU2006 벤치마크와 여러 개의 태스크로 이루어진 swaptions, blacksholes, ferret 등의 PARSEC 벤치마크를 사용하였다. $\hat{v}_{max}(t)$ 측정 시 사용한 방법과 동일하게, SPEC CPU2006에 속하는 벤치마크들은 동일하게 16개를 복사하여 시스템에 부가하였고 PARSEC에 속하는 벤

치마크들은 16개의 태스크를 한 개의 응용프로그램으로 생성 후 시스템에 부가하였다.

표 10은 벤치마크 수행 결과를 제안된 기법을 적용했을 때와 적용하지 않았을 때를 비교한 것이다. SPEC CPU2006에 속하는 벤치마크들은 서

표 10. 벤치마크로 측정한 런타임 오버헤드

Benchmark		**Elapsed time(s)**	**Std. Dev.**	**Overhead (%)**
bzip2	ORG	889	212	-
	SVR	888	204	-0.1
bwaves	ORG	279	69	-
	SVR	282	55	0.92
mcf	ORG	488	56	-
	SVR	491	56	0.77
swaptions	ORG	68		-
	SVR	69		0.82
blackscholes	ORG	667		-
	SVR	655		-1.7
ferret	ORG	75		-
	SVR	76		1.34

로 동일한 태스크를 부가하였기에 그들간의 완료 시간 표준 편차를 측정하였고, PARSEC에 속하는 벤치마크들은 서로 다른 동작을 수행하는 16개의 태스크들이기에 이들의 표준 편차는 기입하지 않았다. 표에서 볼 수 있듯이 평균 런타임 오버헤드는 0.34%에 불과하였다. 또한, 한 개의 태스크로 이루어진 workload와 여러 개의 태스크로 이루어진 workload에서, 런타임 오버헤드 차이가 나타나지 않았다.

이렇게 추가되거나 변경된 소프트웨어 모듈로 인한 런타임 오버헤드가 작은 이유는 다음과 같다. OLS regression 수행 시, 20 tick 동안의 IPT(Instruction per tick) 데이터를 regression한다. 이때, 20 tick에 해당하는 데이터를 매번 측정하지 않고, 매 tick마다 최근 1개의 IPT 데이터를 추가하고 가장 오래된 데이터는 버린다. 이러한 점이 큰 오버헤드로 작용하지 않았다. 그리고 모든 태스크의 20 tick 동안의 IPT 데이터를 저장해야 하는, 메모리 사용측면에서의 오버헤드가 있을 수 있다. 본 연구에서는 이에 필요한 데이터 size를 최소화 하였고, 실제 size는 전체적으로 수 KB에 불과하였다.

다음으로 부하분산 측면에서 살펴본다. 기존 Linaro 스케줄링 프레임웍 대비, 제안된 스케줄링 기법은 merge-sort 및 태스크 그루핑 등이 추가되었다. 이는 실제 태스크를 이주시키기 전에 취하는 단순 연산 작업이며, 전체론적인 관점에서 보면 차지하는 시간은 1% 보다 작은 값이다.

제 6 장 결 론

본 논문은 임베디드 시스템에서 그 사용이 늘어가고 있는, 비대칭 멀티코어 아키텍처의 하드웨어적 동작방식에 따른 최적화된 태스크 스케줄링 기법을 제안하였다. 구체적으로, 비대칭 멀티코어 아키텍처의 코어 타입 선택 방식과 전체 코어 사용 방식에 사용되는 스케줄링 기법의 문제점을 지적하고, 각각에 대한 최적화 방법을 제시하였다.

첫째, 코어 타입 선택 방식에 최적화된 스케줄링 기법을 제안하였다. 이를 위하여 먼저, Linux kernel이 비대칭 멀티코어 아키텍처를 위하여 제공하는 DVFS 정책을 분석하였다. 분석된 내용을 토대로, 코어의 사용률에 의해서 변경되는 동작 주파수와 코어 타입 선택 과정을 하나의 상태도로 간략하게 표현하였다. 이 상태도를 기반으로, 태스크가 코어에 할당될 때 사용률을 추정하는 메커니즘을 Linux kernel의 부하분산 정책에 반영하였다. 이 메커니즘으로 인하여, 현재 운용되는 코어 타입을 직접적으로 알지 않아도, 에너지 효율적으로 태스크를 코어에 할당 할 수 있는 능력을 Linux kernel이 갖추게 되었다.

둘째, 전체 코어 사용 방식을 위한 공정할당 스케줄링 기법을 제안하였다. 이를 위하여 먼저, Linux kernel의 CFS를 확장하여 전체 코어 사용 방식에 맞게 구성한 Linaro 스케줄링 프레임웍을 분석하였다. 분석한 결과, 동일한 태스크들도 코어의 동작 주파수 혹은 코어 타입에 의존적으로 수행한 정도가 다르게 나타날 수 있으나, 이러한 점이 CFS에 반영되지 않음을 알 수 있었다. CFS는 태스크들의 상대적 진척도를 virtual runtime을 통하여 나타낸다. 따라서 본 연구에서는 성능 비대칭성을 반

영한 SVR(scaled virtual runtime)을 정의하고, 이를 CFS의 per-core 스케줄러에 반영하였다. 또한, 기존의 CFS는 모든 태스크들의 상대적 진척도를 비슷하게 유지하기 위하여 가중치를 기반으로 부하분산을 수행함을 분석을 통하여 알았다. 본 연구에서는 이를 개선하여 부하분산 시 모든 태스크들의 SVR(scaled virtual runtime)이 비슷하게 유지하는 알고리즘을 제안하였다. 구체적으로, 동일 클러스터 내부의 태스크들간 최대 SVR 값 차이가 작은 상수 값 이내로 제한되게 하였다.

비대칭 멀티코어 아키텍처의 두 가지 하드웨어적 동작 방식에, 제안된 스케줄링 기법들이 효용성이 있다는 것을 검증하였다. 이를 위해서 본 학위논문에서는 ARM사의 빅리틀 아키텍처를 대상시스템으로 하였다. 각각의 기법들을 빅리틀 아키텍처를 기반으로 하는 실제 상용제품들에 구현하였다. 구체적으로, 코어 타입 선택 방식에 최적화된 스케줄링 기법은 Galaxy S4 Android 스마트폰에 구현 되었다. 실험을 통하여 확인한 결과, 제안된 메커니즘에 의하여 빅리틀 아키텍처의 Linux CFS는 태스크를 코어에 할당할 때 코어의 사용률을 예측함으로써, 에너지 소비를 최적화 시킴을 알았다. CPU-intensive한 workload를 사용하여 실험 시, 기존 대비 최대 11.35%의 에너지 소비가 감소하였다. 또한 Android 응용 프로그램인 소프트웨어 디코더 실험 시, 기존 대비 동일한 QoS를 유지하면서 7.35% 에너지 소비가 감소하였다.

전체 코어 사용 방식을 위한 공정할당 스케줄링 기법은 ARM사의 Versatile Express TC2 board에 구현하였다. 한 개의 태스크로 이루어진 벤치마크를 여러 개 복사해서 시스템에 부가한 결과와, 여러 개의 태스크로 이루어진 벤치마크를 실험한 결과 모두에서, 클러스터 내부의 태스크간 최대 SVR 차이가 상수 값 이내로 제한됨을 확인하였다. 또한 동일한 CPU-intensive 태스크를 여러 개 시스템에 부가한 결과, 완료시간

편차가 기존 대비 56% 줄어 들었다. 이는 태스크들의 상대적인 진척도가 기존 대비 훨씬 더 비슷하게 유지됨을 직관적으로 보여주고 있다.

본 연구를 토대로, 다음과 같은 방법으로 비대칭 멀티코어 아키텍처에 최적화된 스케줄링 기법 연구를 확장할 수 있다. 첫째, 빅리틀 아키텍처가 제공하는 코어 타입 선택 방식에 최적화된 스케줄링 기법을 구현하고자, 본 연구에서는 태스크의 사용률 추정기를 사용하였다. 실험을 통하여 확인했을 때, 두 가지 제안된 추정기 중 태스크가 코어에 할당될 경우 그 사용률을 예측하는 추정기가 에너지 소비 감소 효과가 더 큼을 알았다. 따라서 이 추정기를 더욱 개선할 경우, 더 큰 에너지 소비 감소 효과를 얻을 수 있을 것으로 예측된다.

둘째, 제안된 공정할당 스케줄링 기법은 성능저하를 최소화 하고자, 기존의 Linaro 스케줄링 프레임웍의 클러스터간 태스크 이주정책을 그대로 사용하였다. 그리고 클러스터 내부의 태스크간 최대 SVR 값 차이를 상수 값 이내로 제한함으로써, 시스템 전체적으로는 최대 SVR 값 차이를 매우 작은 값으로 유지하였다. 향 후, 시스템 전체적으로 최대 SVR 값 차이를 작은 상수 값 이내로 제한시키는 연구가 필요하다. 이때, 클러스터간 태스크 이주정책에 대한 수정은 반드시 필요하다. 이에 대한 성능저하 최소화 기법이 연구된다면, 가장 완벽한 공정할당 스케줄링 기법의 결실을 얻을 것으로 기대된다.

참고 문헌

[1] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn, “Operating system support for overlapping-ISA heterogeneous multicore architectures,” in In Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA), Jan. 2010, pp.1–12.

[2] K. V. Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, “Fairness-aware scheduling on single-ISA heterogeneous multi-cores,” in Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT) Sep., 2013, pp. 177-188.

[3] S. Huh, J. Yoo, M. Kim, and S. Hong, “Providing fair share scheduling on multicore cloud servers via virtual runtime-based task migration algorithm,” in IEEE 32nd International Conference on Distributed Computing Systems (ICDCS), Jun. 2012, pp. 606-614.

[4] M. Kim, K. Kim, J. Geraci, and S. Hong, “Utilization-aware load balancing for the energy efficient operation of the big.LITTLE processor,” in In Proceedings of the Conference on Design, Automation & Test in Europe (DATE), Mar. 2014, pp. 1-4.

[5] ARM. (2015) High-performance applications processing for mobile and enterprise markets. [Online]. Available: http://www.arm.com/products/processors/cortex-a/index.php

[6] ARM. (2015) big.LITTLE technology, hardware requirements. [Online]. Available: http://www.arm.com/products/processors/technologies/biglittleprocessing.php

[7] T. Li, D. Baumberger, and S. Hahn, “Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin,” ACM SIGPLAN Notice, vol. 44, no. 4, pp. 65-74, Apr. 2009.

[8] A. K. Parekh and R. G. Gallagher, “A generalized processor sharing approach to flow control in integrated services networks: the multiple node case,” IEEE/ACM Transactions on Networking (TON), vol. 2, no. 2, pp. 137-150, Apr. 1994.

[9] ARM. (2012) Coretile express technical reference manual. [Online]. Available: http://www.arm.com/files/pdf/DDI0503B v2p ca15 a7 reference manual.pdf

[10] B. Caprita, W. C. Chan, J. Nieh, C. Stein, and H. Zheng, “Group ratio round-robin: O (1) proportional share scheduling for uniprocessor and multiprocessor systems,” in In Proceedings of the USENIX Annual Technical Conference, Apr. 2005, pp. 337-352.

[11] A. Chandra, M. Adler, P. Goyal, and P. Shenoy, “Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors,” in In Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation, vol. 4, Oct. 2000.

[12] A. Srinivasan and J. H. Anderson, “Fair scheduling of dynamic task systems on multiprocessors,” Journal of Systems and Software, vol. 77, no. 1, pp. 67-80, Jul. 2005.

[13] C. Kolivas. (2010) BFS scheduler. [Online]. Available: http://ck.kolivas.org/patches/bfs/bfs-faq.txt

[14] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, “Quincy: fair scheduling for distributed computing clusters,” in In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP), Oct. 2009, pp. 261-276.

[15] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, “Choosy: Maxmin fair sharing for datacenter jobs with constraints,” in In Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys), Apr. 2013, pp. 365-378.

[16] B. Caprita, J. Nieh, and C. Stein, “Grouped distributed queues: distributed queue, proportional share multiprocessor scheduling,” in In Proceedings of the 25th Annual ACM symposium on Principles of Distributed Computing (PODC), Jul. 2006, pp. 72-81.

[17] C. Shih, J. Wei, S. Hung, J. Chen, and N. Chang, “Fairness scheduler for virtual machines on heterogonous multi-core platforms,” ACM SIGAPP Applied Computing Review, vol. 13, no. 1, pp. 28-40, Mar. 2013.

[18] S. Hofmeyr, J. A. Colmenares, C. Iancu, and J. Kubiatowicz, “Juggle: proactive load balancing on multicore computers,” in In Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC), Jun. 2011, pp. 3-14.

[19] S. Hofmeyr, C. Iancu, and F. Blagojevic, “Load balancing on speed,” ACM SIGPLAN Notice, vol. 45, no. 5, pp. 147-158, May 2010.

[20] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, “Efficient operating system scheduling for performance-asymmetric multi-core architectures,” in In Proceedings of Supercomputing: the Conference on High Performance Computing Networking, Storage and Analysis (SC), Nov. 2007, pp. 1-11.

[21] V. Kazempour, A. Kamali, and A. Fedorova, “AASH: an asymmetry aware scheduler for hypervisors,” in In Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Mar. 2010, pp. 85-96.

[22] Y. Kwon, C. Kim, S. Maeng, and J. Huh, “Virtualizing performance asymmetric multi-core systems,” in In Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA), Jun. 2011, pp. 45-56.

[23] S. Huh, J. Yoo, and S. Hong, “Improving interactivity via VT-CFS and framework-assisted task characterization for Linux/Android smartphones,” in Embedded and Real-Time Computing Systems and Applications (RTCSA), Aug. 2012, pp. 250-259.

[24] S. Huh, J. Yoo, and S. Hong, “Cross-layer resource control and scheduling for improving interactivity in Android,” Software: Practice and Experience, 2014.

[25] PARSEC benchmark suites. [Online]. Available: http://parsec.cs.princeton.edu/index.htm

[26] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: characterization and architectural implications,” in In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT), Oct. 2008, pp. 72-81.

[27] “Advances in big.little technology for power and energy savings,” [Online]. Available: http://www.thinkbiglittle.com/

[28] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, “Singleisa heterogeneous multi-core architectures: The potential for processor power reduction,” in Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, 2003.

[29] M. Hilzinger. Con kolivas introduces new bfs scheduler. http://www.linux-magazine.com/Online/News/Con-Kolivas-Introduces - New-BFS-Scheduler/. (2009)

[30] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, “Hass: a scheduler for heterogeneous multicore systems,” SIGOPS Oper. Syst. Rev., vol. 43, no. 2, pp. 66-75, Apr. 2009.

[31] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, “Scheduling heterogeneous multi-cores through performance impact estimation (pie),” in Computer Architecture, 2012 39th Annual International Symposium on, pp. 213-224.

[32] L. Sawalha, S. Wolff, M. Tull, and R. Barnes, “Phase-guided scheduling on singleisa heterogeneous multicore processors,” in

Digital System Design, 2011 14th Euromicro Conference on, 2011, pp. 736-745.

[33] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov, “A comprehensive scheduler for asymmetric multicore systems,” in EuroSys, 2010, pp. 139-152.

[34] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, “Pack & cap: adaptive dvfs and thread packing under power caps,” in Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, 2011, pp. 175-185.

[35] X. Li, G. Yan, Y. Han, and X. Li, “Smartcap: User experience-oriented power adaptation for smartphone`s application processor,” in Design, Automation Test in Europe Conference Exhibition, 2013, pp. 57-60.

[36] “CPU frequency and voltage scaling code in the linux(tm) kernel,” [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

[37] “Corelink cci-400 cache coherent interconnect,” [Online]. Available: http://www.arm.com/products/system-ip/interconnect/corelink-cci-400.php

[38] M. Poirier, “In kernel switcher,” [Online]. Available: http://events.linuxfoundation.org/images/stories/slides/elc2013 poirier.pdf

[39] “This is the cfs scheduler,” May 2007. [Online]. Available: http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt

[40] Thannirmalai Somu Muthukaruppan, Mihai Pricopi, Vanchinathan Venkataramani, Tulika Mitra, Sanjay Vishin, “Hierarchical power management for asymmetric multi-core in dark silicon era,” in Proceedings of the Design Automation Conference, 2013.

[41] Linaro, [Online]. Available: https://www.linaro.org/

[42] Paul Turner. 2013. Per-entity load tracking. [Online]. Available: https://lwn.net/Articles/531853/. 2013

[43] Paramveer S. Dhillon, Dean P. Foster, Sham M. Kakade, Lyle H. Ungar, “A Risk Comparison of Ordinary Least Squares vs Ridge Regression,” Journal of Machine Learning Research, 2013

[44] Myungsun Kim, Soonhyun Noh, Sungju Huh, and S. Hong. 2015. “Fair-share Scheduling for Performance-asymmetric Multicore Architecture via Scaled Virtual Runtime,” In Proceedings of the 21st international conference Embedded and Real-Time Computing Systems and Applications (RTCSA). 2015.

[45] SPEC CPU2006 benchmark suites. [Online]. Available: https://www.spec.org/cpu2006

[46] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. SIGARCH Comput. Archit. News 34, 4 (September 2006), 1-17. DOI=http://dx.doi.org/10.1145/1186736.1186737

Abstract

Fair-share and Energy-efficient Scheduling in Performance-asymmetric Multicore Architecture

Myungsun Kim
School of Electrical Engineering and Computer Science
The Graduate School
Seoul National University

As users begin to demand applications with superior user experience and high service quality, asymmetric multicore processors are increasingly adopted in embedded systems due to their architectural benefits in improved performance and power savings. Those asymmetric multicore processors which are particularly designed for embedded systems often take the form of performance-asymmetric and single-ISA(Instruction Set Architecture) multicore architecture. Such architecture consists of high-performance cores and energy-efficient cores. By assigning compute-intensive workloads to high-performance cores while others to energy-efficient cores, we can achieve a performance goal within a limited power and area budget.

There are two distinct modes in operation for this architecture: i) *Switching* and ii)

Global. In the *Switching* mode, each high-performance core is paired with an energy-efficient core and only one core per high-performance/energy-efficient pair can be running at any given time. In the *Global* mode, all cores are available for execution at any time and thus the architecture can exhibit the most flexibility in scheduling. ARM's big.LITTLE architecture is a representative example of such performance-asymmetric multicore architecture which enables a varying number of big and little cores to be utilized at the same time. Linux kernel is widely used for this architecture and its CFS (completely fair scheduler) supports both *Switching* mode and *Global* mode.

Unfortunately, the present CFS for each operating mode has several limitations. First, the CFS with *Switching* mode support neither distinguishes between the core types nor considers how heavily a core is being used when assigning tasks. Therefore, it runs the chance of running a task unnecessarily on a high-frequency core, unnecessarily increasing the core frequency or causing unneeded little cores to big cores transitions. These all cause the processor to consume more energy than necessary.

Second, the present CFS with *Global* mode support falls short of expectations when it comes to fair-share scheduling. It simply lets runnable tasks share physical CPU time in proportion to their weights. In reality, in an asymmetric multicore system, computing power significantly varies between a big core and a little core. Thus, the physical CPU time given to a task should be scaled according to the computing power of the hosting core. In addition, the CFS tries to achieve fair-share scheduling by minimizing the virtual runtime differences among all runnable tasks. It achieves this goal in per-core scheduling, whereas in multicore scheduling, it cannot guarantee fair-share scheduling as CFS's weight-based load balancing has

nothing to do with minimizing the virtual runtime differences.

To overcome abovementioned limitations, this thesis proposed two optimization methods for scheduling in performance-asymmetric multicore architecture. Especially, we took ARM's big.LITTLE architecture as a target system and presented solution approaches. First, we proposed a utilization-aware load balancing mechanism to provide the *Switching* mode with energy-efficient operation. To do so, we carefully analyzed the power management scheme of the big.LITTLE processor's port of Linux kernel and derived its state diagram representations. Based on our formulation, we incorporated the processor utilization factor into the Linux kernel's load balancing algorithm. As a result, our mechanism improved the Linux kernel's ability to assign tasks to cores in an energy-efficient manner without having to make it directly aware of the available core types.

Second, we proposed a solution approach for the fair-share scheduling in the *Global* mode. It uses scaled CPU time which reflects the relative performance of cores with respect to operating frequencies as well as different core types. Since it is a modified version of CFS, each task gets scaled CPU time in proportional to its weight. Thus, the weighted-based virtual runtime of original CFS is extended to the weight-based SVR (scaled virtual runtime) by adopting the scaled CPU time. We also presented SVR-based load balancing algorithm. It periodically makes tasks with larger SVRs run more slowly since they are allocated to the core with heavier load in the following balancing period. As a result, It bounds the SVR difference between any pair of tasks in the same cluster by a constant.

To evaluate the effectiveness of the proposed approaches, we have implemented both solutions on top of commercial products. We have implemented the utilization-aware load balancing mechanism into Galaxy S4. Experimental results

show that it can reduce energy consumption by 11.35% compared to the original Linux CFS. Our method's energy efficiency and computation performance depend on the utilization estimator, so better estimators could produce greater gains. Complexity of any estimator, however, must be simple enough so as to not burden the scheduler with its compute time, This trade-off between estimator complexity of implementation and the benefits provided by the increased complexity make further research into utilization estimators a promising area of future work. We have also implemented the solution approach for fair-share scheduling into ARM's Versatile TC2 board. Experimental results show that the maximum SVR difference in our solution was bounded by a constant while it diverges indefinitely in the original CFS. The standard deviation of finish time of multiple identical tasks under our approach is smaller by 56% than the original CFS. These results clearly demonstrate that our proposed approach can provide enhanced fairness in performance-asymmetric multicore architecture.

Keywords : Performance-asymmetric multicore, tsk scheduling, load balancing, low power scheduling, big.LITTLE architecture

Student Number : 2011-30218